Pushdown Module Checking with Imperfect Information

Benjamin Aminof, Axel Legay, Aniello Murano, Olivier Serre, Moshe Vardi

Pushdown Module Checking with Imperfect Information[☆]

Benjamin Aminof[a], Axel Legay[b], Aniello Murano[c], Olivier Serre[d], Moshe Y. Vardi[e]

[a]*Hebrew University, Jerusalem 91904, Israel.*
[b]*INRIA/IRISA, Rennes.*
[c]*Università di Napoli "Federico II".*
[d]*LIAFA, CNRS & Université Paris VII, France.*
[e]*Rice University, TX 77251-1892, USA*

Abstract

The model checking problem for finite-state open systems (*module checking*) has been extensively studied in the literature, both in the context of environments with perfect and imperfect information about the system. Recently, the perfect information case has been extended to infinite-state systems (*pushdown module checking*). In this paper, we extend pushdown module checking to the imperfect information setting; i.e., to the case where the environment has only a partial view of the system's control states and pushdown store content. We study the complexity of this problem with respect to the branching-time temporal logics CTL, CTL^* and the propositional μ-calculus. We show that pushdown module checking, which is by itself harder than pushdown model checking, becomes undecidable when the environment has imperfect information.

We also show that undecidability relies on hiding information about the pushdown store. Indeed, we prove that with imperfect information about the control states, but a visible pushdown store, the problem is decidable and its complexity is 2Exptime-complete for CTL and the propositional μ-calculus, and 3Exptime-complete for CTL^*.

[☆]This work is based on the papers [3] appeared in CONCUR 2007 and [5] appeared in IFIP-TCS 2008.

Email addresses: benj@cs.huji.ac.il (Benjamin Aminof), alegay@irisa.fr (Axel Legay), murano@na.infn.it (Aniello Murano), Olivier.Serre@liafa.jussieu.fr (Olivier Serre), vardi@cs.rice.edu (Moshe Y. Vardi)

Keywords: Model checking, open systems, infinite-state systems, branching-time temporal-logics, games.

1. Introduction

In system modeling we distinguish between *closed* and *open* systems [21]. In a closed system all the nondeterministic choices are internal, and resolved by the system. In an open system there are also external nondeterministic choices, which are resolved by the environment [22]. In order to check whether a closed system satisfies a required property, we translate the system into some formal model, specify the property with a temporal-logic formula, and check formally that the model satisfies the formula. Hence, the name *model checking* for the verification methods derived from this viewpoint ([14, 35]).

In [27, 30], Kupferman, Vardi, and Wolper studied open finite-state systems. In their framework, the open finite-state system is described by a labeled state-transition graph called a *module*, whose set of states is partitioned into a set of *system states* (where the system makes a transition) and a set of *environment states* (where the environment makes a transition). Given a module $\mathcal{M}$ describing the system to be verified, and a temporal logic formula φ specifying the desired behavior of the system, the problem of model checking a module, called *module checking*, asks whether for all possible environments $\mathcal{M}$ satisfies φ. In particular, it might be that the environment does not enable all the external nondeterministic choices. Module checking thus involves not only checking that the full computation tree $\langle T_M, V_M \rangle$ obtained by unwinding $\mathcal{M}$ (which corresponds to the interaction of $\mathcal{M}$ with a maximal environment) satisfies the specification φ, but also that every tree obtained from it by pruning children of environment nodes (this corresponds to the different choices of different environments) satisfy φ.

For example, consider an ATM machine that allows customers to deposit money, withdraw money, check balance, etc. The machine is an open system and an environment for it is a subset of the set of all possible infinite lines of customers, each with his/her own plans. Accordingly, there are many different possible environments to consider. It is shown in [27, 30, 17] that for formulas in branching time temporal logics, module checking open finite-state systems is exponentially harder than model checking closed finite-state systems.

In [26] module checking has been extended to a setting where the envi-

ronment has *imperfect information*[1] about the state of the system (see also [13, 12], for related work regarding imperfect information). In this setting, every state of the module is a composition of *visible* and *invisible* variables, where the latter are hidden from the environment. While a composition of a module $\mathcal{M}$ with an environment with perfect information corresponds to arbitrary disabling of transitions in $\mathcal{M}$, the composition of $\mathcal{M}$ with an environment with imperfect information is such that whenever two computations of the system differ only in the values of internal variables along them, the disabling of transitions along them coincide.

For example, in the above ATM machine, a person does not know, before he asks for money, whether or not the ATM has run out of paper for printing receipts. Thus, the possible behaviors of the environment are independent of this missing information. Given an open system $\mathcal{M}$ with a partition of $\mathcal{M}$'s variables into visible and invisible ones, and a temporal logic formula φ, the module-checking problem with imperfect information asks whether φ is satisfied by all trees obtained by pruning children of environment nodes from $\langle T_M, V_M \rangle$, according to environments whose nondeterministic choices are independent of the invisible variables. One of the results shown in [26] is that *CTL* module checking with imperfect information is EXPTIME-complete.

In recent years, model checking of pushdown systems has received a lot of attention (see for example [37, 38, 8, 16]), largely due to the ability of pushdown systems to capture the flow of procedure calls and returns in programs [1, 2, 4]. Recently, [9, 10] extended these techniques by introducing open pushdown systems (with perfect information) that interact with their environment. It is shown in [9, 10] that *CTL pushdown module checking* is 2EXPTIME-complete and thus much harder than pushdown model checking.

Consider again the example of the ATM machine, where the information regarding the presence of printing paper is invisible to the customers. Suppose also that the ATM machine shows advertisements, and that it works under the constraint that the number of advertisements the customer must view, before the card can be taken out of the machine, is equal to the number of operations the customer performed. The described machine can be modeled as an open pushdown system $\mathcal{M}$ where control states take care of the operation performed by the ATM (interacting with customers), and the

[1] In the literature, the term *incomplete information* is sometimes used to refer to what we call imperfect information.

pushdown store is used to keep track of the advertisements that remain to be shown. Now, suppose that we want to verify that in all possible environments, it is always possible for an inserted card to be ejected. This requirement can be modeled by the *CTL* formula $\varphi = AG(\textbf{insert-card} \rightarrow EF\textbf{eject-card})$. Since the presence of printing paper is invisible to the customers, we have imperfect information about the control states of the module. If we allow the ATM to push, after each operation the customer makes, an invisible number (possibly zero) of pending advertisements, then we also have invisible information in the pushdown store.

In this work we extend pushdown module checking by considering environments with imperfect information about the system's control state and pushdown store content. To this aim, we first have to define how a pushdown system keeps part of its internal configuration invisible to the environment and another part visible. In [34], a *private pushdown store automata* is defined to be a Turing machine with two tapes: a read only public (visible) one-way input tape, and a possibly private (invisible) work tape, simulating a pushdown store. Unfortunately, their definition is not suitable for our purpose as it allows for only two levels of information hiding: either the pushdown store and control state are completely visible, or completely invisible. The definition we use instead is an extension of the idea used for finite-state systems. Like in the finite case, we assume the control states are assignments to boolean *control variables*, some of which are visible and some of which are invisible. Similarly, symbols of the pushdown store are assignments to boolean visible and invisible *pushdown store variables*.

In [26], each state is partitioned into input, output, and invisible variables, where the environment supplies the input variables, and the system supplies the output and invisible variables. This idea works well for finite state-systems but not when we have to deal with imperfect information about the pushdown store. Note that a symbol pushed now influences the computation much later, when it becomes the top of the pushdown store. Indeed, asking the environment to supply as input part of each symbol in the pushdown, is asking it to intimately participate in the internals of the computation, which is less natural. We find it more natural to think of the environment as choosing the possible transitions at certain points of the computation. For example, if the environment supplies the current reading of a physical sensor, we think of it as disabling all the transitions that are irrelevant for this reading. Thus, we model an open pushdown system with imperfect information by partitioning configurations into system and environment configurations,

and also partitioning states and pushdown store symbols into visible and invisible variables, combining features from both [27] and [26].

We study the pushdown module-checking problem with imperfect information, with respect to the branching-time logics CTL, CTL^* and the propositional μ-calculus. We show that the problem is undecidable in the general case, and that undecidability relies on hiding information about the pushdown store. In other words, the problem becomes decidable with imperfect information about the internal control states, but a visible pushdown store. We derive both the undecidability and the decidability results using an automata theoretic approach. For the undecidability, we give a reduction of the universality problem of nondeterministic pushdown automata on finite words, which is undecidable [23], to the CTL pushdown module-checking problem with imperfect information. In order to derive the decidability results, we reduce the pushdown module-checking problem with imperfect state information to the emptiness problem of a new automata model that we call *semi-alternating pushdown tree automata*. These are alternating pushdown tree automata [25] (see also [31]), where the universality is not allowed on the pushdown store content. That is, two copies of the automaton that read the same input, from two configurations that have the same top of pushdown store, must push the same value into the pushdown store.

We consider two types of acceptance conditions for semi-alternating pushdown tree automata, one is the Büchi condition which is suitable for handling CTL, and the other is the parity condition which is suitable for CTL^* and μ-calculus. We show that, for both acceptance conditions, semi-alternating pushdown tree automata can be translated to equivalent nondeterministic pushdown tree automata (with the same acceptance condition), for which the emptiness problem is decidable in exponential time [25]. It is important to note that alternating pushdown automata, in contrast to the semi-alternating ones, are *not* equivalent to nondeterministic pushdown automata. Indeed, since the emptiness problem of the intersection of two context free languages is undecidable [23], the emptiness problem of alternating pushdown automata is undecidable already in the case of finite words.

Overall, based on the above translations, we are able to show that the pushdown module checking problem with imperfect state information is decidable and 2EXPTIME-complete for CTL and propositional μ-calculus specifications, and is 3EXPTIME-complete for CTL^* specifications. Hence, in all cases, it is not harder than perfect information pushdown module checking.

2. Preliminaries

2.1. Trees

Let Υ be a set. An Υ-*tree* is a prefix closed subset $T \subseteq \Upsilon^*$. The elements of T are called *nodes* and the empty word ε is the *root* of T. For $v \in T$, the set of *children* of v (in T) is $child(T, v) = \{v \cdot x \in T \mid x \in \Upsilon\}$. Given a node $v = u \cdot x$, with $u \in \Upsilon^*$ and $x \in \Upsilon$, we define $last(v)$ to be x. We also say that v *corresponds* to x. The complete Υ-tree is the tree Υ^*. For $v \in T$, a (full) path π of T from v is a *minimal* set $\pi \subseteq T$ such that $v \in \pi$ and for each $v' \in \pi$ such that $child(T, v') \neq \emptyset$, there is exactly one node in $child(T, v')$ belonging to π. Note that every infinite word $w \in \Upsilon^\omega$ can be thought of as an infinite path in the tree Υ^*, namely the path containing all the finite prefixes of w. For an alphabet Σ, a Σ-labeled Υ-tree is a pair $\langle T, V \rangle$ where T is an $\Upsilon-$tree and $V : T \to \Sigma$ maps each node of T to a symbol in Σ.

2.2. The propositional μ-Calculus

The μ-calculus is a propositional modal logic augmented with least and greatest fixpoint operators. We consider a μ-calculus where formulas are constructed from Boolean propositions with Boolean connectives, the temporal operators EX ("exists next") and AX ("for all next"), as well as least (μ) and greatest (ν) fixpoint operators. We assume that μ-calculus formulas are written in positive normal form (negation only applied to atomic propositions).

Formally, let AP and Var be finite and pairwise disjoint sets of *atomic propositions* and *propositional variables*. The set of *μ–calculus* formulas over AP and Var is the smallest set such that

- true and false are formulas;
- p and $\neg p$, for $p \in \mathsf{Prop}$, are formulas;
- $x \in \mathsf{Var}$ is a formula;
- $\varphi_1 \vee \varphi_2$ and $\varphi_1 \wedge \varphi_2$ are formulas if φ_1 and φ_2 are formulas;
- $AX\varphi$ and $EX\varphi$ are formulas if φ is a formula;
- $\mu y.\varphi(y)$ and $\nu y.\varphi(y)$ are formulas if y is a propositional variable and $\varphi(y)$ is a formula containing y as a free variable.

Observe that we use positive normal form, i.e., negation is applied only to atomic propositions.

We call μ and ν *fixpoint operators* and use λ to denote a fixpoint operator μ or ν. A propositional variable y occurs *free* in a formula if it is not in the scope of a fixpoint operator λy, and *bounded* otherwise. Note that y may occur both bounded and free in a formula. A *sentence* is a formula that contains no free variables. For a formula $\lambda y.\varphi(y)$, we write $\varphi(\lambda y.\varphi(y))$ to denote the formula that is obtained by one-step unfolding, i.e., replacing each free occurrence of y in φ with $\lambda y.\varphi(y)$.

The closure $cl(\varphi)$ of a μ-calculus sentence φ is the smallest set of μ-calculus formulas that contains φ and is closed under sub-formulas (that is, if ψ is in the closure, then so do all its sub-formulas that are sentences) and fixpoint applications (that is, if $\lambda y.\varphi(y)$ is in the closure, then so is $\varphi(\lambda y.\varphi(y))$). For every μ-calculus formula φ, the number of elements in $cl(\varphi)$ is linear in the length of φ. Accordingly, we define the size $|\varphi|$ of φ to be the number of elements in $cl(\varphi)$. A μ-calculus formula is *guarded* if for every variable y, all the occurrences of y that are in a scope of a fixpoint modality λ are also in a scope of a modality AX or EX that is itself in the scope of λ. Thus, a μ-calculus sentence is guarded if for all $y \in Var$, all the occurrences of y are in the scope of a next modality. Given a μ-calculus formula, it is always possible to construct in linear time an equivalent guarded formula (c.f.,[29, 6, 7]).

The semantics of the μ–calculus is defined with respect to a *Kripke structure* $K = \langle AP, W, R, w_0, L\rangle$, where AP is a set of atomic propositions, W is a finite set of states, $R \subseteq W \times W$ is a transition relation that must be total (i.e., for every $w \in W$ there exists $w' \in W$ such that $(w, w') \in R$), w_0 is an initial state, and $L : W \to 2^{AP}$ maps each state to the set of atomic propositions true in that state. If $(w, w') \in R$, we say that w' is a *successor* of w. For each $w \in W$, we denote by $succ(w)$ the set of w's successors. A *path* in K is an infinite sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, $(w_i, w_{i+1}) \in R$. We denote the suffix $w_i, w_{i+1}, \ldots$ of π by π^i. We define the size $|K|$ of K as $|W| + |R|$.

Informally, a formula $EX\varphi$ holds at a state w of a Kripke structure K if φ holds at least in one successor of w. Dually, the formula $AX\varphi$ holds in a state w of a Kripke structure K if φ holds in all successors of w. Readers not familiar with fixpoints might want to look at [24, 36, 11] for instructive examples and explanations of the semantics of the μ-calculus. To formalize the semantics, we introduce valuations that allow to associate sets of points to vari-

ables. Given a Kripke structure $K = \langle AP, W, R, w_0, L\rangle$ and a set $\{y_1, \ldots, y_n\}$ of propositional variables in Var, a *valuation* $\mathcal{V} : \{y_1, \ldots, y_n\} \to 2^W$ is an assignment of subsets of W to the variables $y_1, \ldots, y_n$. For a valuation $\mathcal{V}$, a variable y, and a set $W' \subseteq W$, we denote by $\mathcal{V}[y \leftarrow W']$ the valuation obtained from $\mathcal{V}$ by assigning W' to y. A formula φ with free variables among $y_1, \ldots, y_n$ is interpreted over the structure K as a mapping φ^K from valuations to 2^W, i.e., $\varphi^K(\mathcal{V})$ denotes the set of states that satisfy φ under valuation $\mathcal{V}$. The mapping φ^K is defined inductively as follows:

- $\mathsf{true}^K(\mathcal{V}) = W$ and $\mathsf{false}^K(\mathcal{V}) = \emptyset$;
- for $p \in AP$, we have $p^K(\mathcal{V}) = L^{-1}(p)$ and $(\neg p)^K(\mathcal{V}) = W \setminus L^{-1}(p)$;
- for $y \in \mathsf{Var}$, we have $y^K(\mathcal{V}) = \mathcal{V}(y)$;
- $(\varphi_1 \wedge \varphi_2)^K(\mathcal{V}) = \varphi_1^K(\mathcal{V}) \cap \varphi_2^K(\mathcal{V})$
- $(\varphi_1 \vee \varphi_2)^K(\mathcal{V}) = \varphi_1^K(\mathcal{V}) \cup \varphi_2^K(\mathcal{V})$;
- $(EX\varphi)^K(\mathcal{V}) = \{w \in W : \exists w'.(w, w') \in R \text{ and } w' \in \varphi^K(\mathcal{V})\}$;
- $(AX\varphi)^K(\mathcal{V}) = \{w \in W : \forall w'.\text{if } (w, w') \in R \text{ then } w' \in \varphi^K(\mathcal{V})\}$;
- $(\mu y.\varphi(y))^K(\mathcal{V}) = \bigcap\{W' \subseteq W : \varphi^K(\mathcal{V}[y \leftarrow W']) \subseteq W'\}$;
- $(\nu y.\varphi(y))^K(\mathcal{V}) = \bigcup\{W' \subseteq W : W' \subseteq \varphi^K(\mathcal{V}[y \leftarrow W'])\}$.

Note that no valuation is required for a sentence.

Let $K = \langle AP, W, R, w_0, L\rangle$ be a Kripke structure and φ a sentence. For a state $w \in W$, we say that φ *holds* at w in K, denoted $K, w \models \varphi$, if $w \in \varphi^K(\emptyset)$. K is a *model* of φ (denoted by $K \models \varphi$) if $K, w_0 \models \varphi$. Finally, φ is *satisfiable* if it has a model.

2.3. The Temporal Logics CTL* *and* CTL

The logic *CTL** combines both branching-time and linear-time operators [15]. A path quantifier, either A ("for all paths") or E ("for some path"), can prefix an assertion composed of an arbitrary combination of the linear-time operators X ("next time"), and U ("until"). A *positive normal form CTL** formula is a *CTL** formula in which negations are applied only to atomic

propositions. It can be obtained by pushing negations inward as far as possible, using De Morgan's laws and dualities of quantifiers and temporal connectives. For technical convenience, we use the linear-time operator $\widetilde{U}$ as a dual of the U operator, and write all CTL^* formulas in a positive normal form. There are two types of formulas in CTL^*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. Formally, let AP be a set of atomic proposition names. A CTL^* state formula is either:

- **true**, **false**, p, or $\neg p$, for all $p \in AP$;
- $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$, where φ_1 and φ_2 are CTL^* state formulas;
- $A\psi$ or $E\psi$, where ψ is a CTL^* path formula.

A CTL^* path formula is either:

- A CTL^* state formula;
- $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $X\psi_1$, $\psi_1 \, U \, \psi_2$, or $\psi_1 \, \widetilde{U} \, \psi_2$, where ψ_1 and ψ_2 are CTL^* path formulas.

CTL^* is the set of state formulas generated by the above rules.

We use the following abbreviations in writing formulas:

- $F\psi = \mathbf{true} \, U \, \psi$ ("eventually").
- $G\psi = \mathbf{false} \, \widetilde{U} \, \psi$ ("always").

The logic CTL is a restricted subset of CTL^* in which the temporal operators must be immediately preceded by a path quantifier. Formally, it is the subset of CTL^* obtained by restricting the path formulas to be $X\varphi_1$, $\varphi_1 \, U \, \varphi_2$, or $\varphi_1 \, \widetilde{U} \, \varphi_2$, where φ_1 and φ_2 are CTL state formulas.

The *closure* $cl(\varphi)$ of a CTL^* (CTL) formula φ is the set of all CTL^* (CTL) state sub-formulas of φ (including φ, but excluding **true** and **false**). We define the *size* $|\varphi|$ of φ as the number of elements in $cl(\varphi)$. Note that, even though the number of elements in the closure of a formula can be logarithmic in the length of the formula if there are multiple occurrences of identical sub-formulas, our definition of size is legitimate since it corresponds to the number of nodes in a reduced DAG representation of the formula.

The notation $K, w \models \varphi$ indicates that a CTL^* state formula φ holds at the state w of the Kripke structure K. Similarly, $K, \pi \models \psi$ indicates that a CTL^* path formula ψ holds on a path π of the Kripke structure K. When K is clear from the context, we write $w \models \varphi$ and $\pi \models \psi$. Also, $K \models \varphi$ iff $K, w_0 \models \varphi$. The relation $\models$ is inductively defined as follows.

- For all w, we have $w \models \mathbf{true}$ and $w \not\models \mathbf{false}$.
- $w \models p$ for $p \in AP$ iff $p \in L(w)$.
- $w \models \neg p$ for $p \in AP$ iff $p \notin L(w)$.
- $w \models \varphi_1 \wedge \varphi_2$ iff $w \models \varphi_1$ and $w \models \varphi_2$.
- $w \models \varphi_1 \vee \varphi_2$ iff $w \models \varphi_1$ or $w \models \varphi_2$.
- $w \models A\psi$ iff for every path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, we have $\pi \models \psi$.
- $w \models E\psi$ iff there exists a path $\pi = w_0, w_1, \ldots$, with $w_0 = w$, such that $\pi \models \psi$.
- $\pi \models \varphi$ for a state formula φ, iff $w_0 \models \varphi$ where $\pi = w_0, w_1, \ldots$
- $\pi \models \psi_1 \wedge \psi_2$ iff $\pi \models \psi_1$ and $\pi \models \psi_2$.
- $\pi \models \psi_1 \vee \psi_2$ iff $\pi \models \psi_1$ or $\pi \models \psi_2$.
- $\pi \models X\psi$ iff $\pi^1 \models \psi$.
- $\pi \models \psi_1 \, U \, \psi_2$ iff there exists $i \geq 0$ such that $\pi^i \models \psi_2$ and for all $0 \leq j < i$, we have $\pi^j \models \psi_1$.
- $\pi \models \psi_1 \, \widetilde{U} \, \psi_2$ iff for all $i \geq 0$ such that $\pi^i \not\models \psi_2$, there exists $0 \leq j < i$ such that $\pi^j \models \psi_1$.

Note that $\pi \models \psi_1 \, \widetilde{U} \, \psi_2$ if and only if $\pi \not\models (\neg\psi_1) \, U \, (\neg\psi_2)$. That is, a path π satisfies $\psi_1 \, \widetilde{U} \, \psi_2$ if ψ_2 holds everywhere along π (thus, the U does not reach its eventuality), or if the first occurrence of $\neg\psi_2$ is strictly preceded by an occurrence of ψ_1 (thus, $\neg\psi_1$ is falsified before the eventuality is reached). Another way to understand the $\widetilde{U}$ operator is to interpret $\psi_1 \, \widetilde{U} \, \psi_2$ by "as long as ψ_1 is false, ψ_2 must be true".

2.4. Open Systems

An *open system* is a system that interacts with its environment and whose behavior depends on this interaction. We consider the case where the environment has imperfect information about the system, i.e., when the system has internal variables that are not visible to its environment. We describe such a system by a *module* $\mathcal{M} = \langle AP, W_s, W_e, w_0, R, L, \cong \rangle$, where AP is a finite set of *atomic propositions*, W_s is a set of *system states*, and W_e is a set of *environment states*. We assume $W_s \cap W_e = \emptyset$, and call $W = W_s \cup W_e$ the set of $\mathcal{M}$'s *states*. $w_0 \in W$ is the *initial state*, $R \subseteq W \times W$ is a total *transition relation*, $L : W \to 2^{AP}$ is a labeling function that maps each state of $\mathcal{M}$ to the set of atomic propositions that hold in it, and $\cong$ is an equivalence relation on W.

In order to present a unified definition that is general enough to handle both finite-state and infinite-state systems, we model the fact that the environment has imperfect information about the states of the system by an equivalence relation $\cong$. States that are indistinguishable by the environment, because the difference between them is kept invisible by the system, are equivalent according to $\cong$. We write $[W]$ for the set of equivalence classes of W under $\cong$. Since states in the same equivalence class are indistinguishable by the environment, from the environment's point of view, the states of the system are actually the equivalence classes themselves. The equivalence class $[w]$ of $w \in W$, is called the *visible part* of w, since it is in a sense what the environment "sees" of w. We write $vis(w)$ instead of $[w]$, to emphasize this. Note that we can also do the converse. That is, given a function vis, whose domain is W, we can define the equivalence relation $\cong$ by letting $w \cong w'$ iff $vis(w) = vis(w')$. We can then think of the range of vis as the set of the equivalence classes $[W]$ and associate $[w]$ with the value $vis(w)$.

A module $\mathcal{M}$ is *closed* if $W_e = \emptyset$ (meaning that $\mathcal{M}$ does not interact with any environment) and *open* otherwise. Since the designation of a state as an environment state is obviously known to the environment, we require that for every $w, w' \in W$ such that $w \cong w'$, we have that $w \in W_e$ iff $w' \in W_e$. Also note that if $w \cong w'$, from the environment's point of view, the set of atomic propositions that currently hold in w may just as well be $L(w')$. We therefore define the labeling, as seen by the environment, as a function $visL : [W] \to 2^{2^{AP}}$ that maps the visible part of a state to a set of possible sets of atomic propositions: $visL([u]) = \{L(w) : w \in W \wedge w \cong u\}$. If it is always the case that $w \cong w' \implies L(w) = L(w')$, we say that the atomic propositions are visible.

As for Kripke structures, saying that R is total means that for every $w \in W$ we have that $succ(w) \neq \emptyset$. A *computation* of $\mathcal{M}$ is a sequence $w_0 \cdot w_1 \cdots$ of states, such that for all $i \geq 0$ we have $(w_i, w_{i+1}) \in R$. When the module $\mathcal{M}$ is in a system state w_s, then all successor states are possible next states. On the other hand, when $\mathcal{M}$ is in an environment state w_e, the environment decides, based on the visible parts of each successor of w_e, and of the history of the computation so far, to which of the successor states the computation can proceed, and to which it cannot.

The set of all (maximal) computations of $\mathcal{M}$ starting from the initial state w_0 can be described by an AP-labeled W-tree $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ called a *computation tree*, which is obtained by unwinding $\mathcal{M}$ in the usual way. Each node $v = v_1 \cdots v_k$ of $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ describes the (partial) computation $w_0 \cdot v_1 \cdots v_k$ of $\mathcal{M}$, with the root ε corresponding to w_0. The children of v are exactly all nodes of the form $v_1 \cdots v_k \cdot w$, where w ranges over all the successors of v_k in $\mathcal{M}$. We extend the definition of the vis function to nodes in the natural way. Thus, the visible part of a node v is $vis(v) = vis(v_1) \cdots vis(v_k)$. The labeling $V_{\mathcal{M}}$ of a node v depends on the state it *corresponds* to (its last state), i.e., $V_{\mathcal{M}}(v) = L(last(v))$. Also, if v corresponds to an environment state, we say that v is an *environment node*.

The problem of deciding, for a given temporal logic formula φ, over the set AP of atomic propositions, whether $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ satisfies φ is the usual *model checking problem* (formally denoted $\mathcal{M} \models \varphi$) [14, 35]. In model checking, we only have to consider the computation tree $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$, since the module we want to check is closed and thus its behavior is not affected by the environment. On the other hand, whenever we consider an open module, $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ corresponds to a very specific environment: a maximal environment that never restricts the set of next states. Therefore, when we examine a branching-time specification φ w.r.t. an open module $\mathcal{M}$, the formula φ should hold not only in $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$, but in all the trees obtained by pruning from $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ subtrees whose roots are children (successors) of environment nodes, in accordance with all *possible* environments. It is important to note that in the case of perfect information (i.e., $\cong$ is actually the equality relation), every such pruning corresponds to some environment; however, in the case of imperfect information, only if the pruning is consistent with the partial information available to the environment will the tree correspond to an actual environment. Formally, if two nodes v and v' are indistinguishable, i.e., if $vis(v) = vis(v')$, then a tree in which the subtree rooted at v is pruned, but the one rooted at v' is not pruned, does not correspond to any environ-

ment, and should not be considered. As noted in [26], the fact that given a pruning of $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$, a finite automaton cannot decide if that pruning corresponds to an actual environment or not, is the main source of difficulty in dealing with module checking with imperfect information. Also note that the knowledge-based subset construction that is used to transform games of imperfect information into ones of perfect information (see for example [12]), is not applicable in this context, since in general there is no connection between the satisfiability of a branching time formula on the original structure and its satisfiability on the one obtained by the knowledge-based subset construction.

Recall that whenever $\mathcal{M}$ interacts with an environment ξ, its possible moves from environment states depends on the behavior of ξ. We can think of an environment to $\mathcal{M}$ as a strategy $\xi : [W]^* \to \{\top, \bot\}$ that maps a finite history s of a computation, as seen by the environment, to either $\top$ or $\bot$, meaning that the environment respectively allows or disallows $\mathcal{M}$ to trace s. In other words, if the environment's choices are such that s cannot be a prefix of any computation of M then $\xi(s) = \bot$; otherwise, $\xi(s) = \top$. We say that the tree $\langle [W]^*, \xi \rangle$ maintains the strategy applied by ξ, and we call it a *strategy tree*. We denote by $\mathcal{M} \lhd \xi$ the AP-labeled W-tree induced by the composition of $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ with ξ; that is, the AP-labeled W-tree obtained by pruning from $\langle T_{\mathcal{M}}, V_{\mathcal{M}} \rangle$ subtrees according to ξ. Note that by the definition above, ξ may disable all the children of a node v. Since we usually do not want the environment to completely block the system, we require that at least one child of each node is enabled. In this case, we say that the composition $\mathcal{M} \lhd \xi$ is *deadlock free*.

To see the interaction of $\mathcal{M}$ with ξ, let $v \in T_{\mathcal{M}}$ be an environment node, and $v' \in T_{\mathcal{M}}$ be one of its children. The subtree rooted in v' is pruned iff $\xi(vis(v')) = \bot$. Every two nodes v_1 and v_2 that are indistinguishable according to ξ's imperfect information have $vis(v_1) = vis(v_2)$. Also, recall that the designation of a state as an environment state is based only on the visible part of that state. Thus, if v_1 is a child of an environment node then so is v_2, and either both subtrees with roots v_1 and v_2 are pruned, or both are not. Note that once $\xi(v) = \bot$ for some $v \in [W]^*$, we can ignore $\xi(v \cdot t)$, for all $t \in [W]^*$. Indeed, once the environment disables the transition to a certain node v, it actually disables the transitions to all the nodes in the subtree with root v. We can now formally define the interaction of an open module with an environment with imperfect information. From now on, unless stated differently, we always refer to modules that are open, and environments with

imperfect information. Given a module $\mathcal{M}$, and a strategy tree $\langle [W]^*, \xi \rangle$ for an environment ξ, an AP-labeled W-tree $\langle T, V \rangle$ corresponds to $\mathcal{M} \lhd \xi$ iff the following hold:

- The root of T corresponds to w_0.
- For $v \in T$ with $last(v) \in W_s$, we have $child(T, v) = \{v \cdot w_1, \ldots, v \cdot w_n\}$, where $succ(last(v)) = \{w_1, \ldots, w_n\}$.
- For $v \in T$ with $last(v) \in W_e$, there is a nonempty subset $\{w_1, \ldots, w_k\}$ of $succ(last(v))$ such that $child(T, v) = \{v \cdot w_1, \ldots, v \cdot w_k\}$. Furthermore, for all w in $\{w_1, \ldots, w_k\}$ we have that $\xi(vis(v \cdot w)) = \top$, while for all w in $succ(last(v)) \setminus \{w_1, \ldots, w_k\}$ we have that $\xi(vis(x \cdot w)) = \bot$.
- For every node $v \in T$, we have that $V(v) = L(last(v))$.

For a module $\mathcal{M}$ and a temporal logic formula over the set AP, we say that $\mathcal{M}$ *reactively satisfies* φ, denoted $\mathcal{M} \models_r \varphi$, if $\mathcal{M} \lhd \xi$ satisfy φ, for every environment ξ for which $\mathcal{M} \lhd \xi$ is deadlock free. The problem of deciding whether $\mathcal{M} \models_r \varphi$ is called *module checking*, and was first introduced and studied in [27, 30] for finite-state systems with perfect information. The problem was successively extended to imperfect information in [26]. It has been shown that the complexity of both problems is EXPTIME-complete[2] for CTL, and 2EXPTIME-complete for CTL^*.

3. Definition of the Problem

In this section, we extend the notion of module checking with imperfect information to infinite-state systems induced by *Open Pushdown Systems* (*OPD*).

Definition 1. *An* OPD *is a tuple* $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \mu, Env \rangle$*, where* AP *is a finite set of atomic propositions,* Q *is the set of (*control*)* states*, and* $q_0 \in Q$ *is an* initial state*. We assume that* $Q \subseteq 2^{I \cup H}$ *where* I *and* H *are disjoint finite sets of* visible *and* invisible control variables*, respectively.* Γ

[2] Although the complexity of the perfect and imperfect information cases coincide in the general case, [30, 26] show that when the formula is constant the imperfect information case is exponentially harder.

is a finite pushdown store alphabet, $\natural \notin \Gamma$ *is the* pushdown store bottom symbol*, and we use* $\Gamma_\natural$ *to denote* $\Gamma \cup \{\natural\}$*. We assume that* $\Gamma \subseteq 2^{I_\Gamma \cup H_\Gamma}$ *where* I_Γ *and* H_Γ *are disjoint finite sets of* visible *and* invisible pushdown store variables*, respectively.* $\delta \subseteq (Q \times \Gamma_\natural) \times (Q \times \Gamma_\natural^*)$ *is a finite transition relation, and* $\mu : Q \times \Gamma_\natural \to 2^{AP}$ *is a labeling function.* $Env \subseteq Q \times \Gamma_\natural$ *is used to specify the set of* environment configurations*. The size* $|\mathcal{S}|$ *of* $\mathcal{S}$ *is* $|Q| + |\Gamma| + |\delta|$*, with* $|\delta| = \sum_{((p,\gamma),(q,\beta)) \in \delta, \beta \neq \varepsilon} |\beta| + |\{((p,\gamma),(q,\varepsilon)) \in \delta\}|$.

A *configuration* of $\mathcal{S}$ is a pair (q, α), where q is a control state and $\alpha \in \Gamma^* \cdot \natural$ is a pushdown store content. We write $top(\alpha)$ for the leftmost symbol of α and call it the *top of the pushdown store* α. The *OPD* moves according to the transition relation. Thus, $((p,\gamma),(q,\beta)) \in \delta$ implies that if the *OPD* is in state p and the top of the pushdown store is γ, it can move to state q, pop γ and push β. We assume that if $\natural$ is popped it gets pushed right back, and that it only gets pushed in such cases. Thus, $\natural$ is always present at the bottom of the pushdown store, and nowhere else. Note that we make this assumption also about the various pushdown automata we use later. Also note that the possible moves of the system, the labeling function, and the designation of configurations as environment configurations, are all dependent only on the current control state and the top of the pushdown store.

For a control state $q \in Q$, the visible part of q is $vis(q) = q \cap I$. For a pushdown store symbol $\gamma \in \Gamma$, if $\gamma \subseteq H_\Gamma$ and $\gamma \neq \emptyset$ we set $vis(\gamma) = \varepsilon$, otherwise we set $vis(\gamma) = \gamma \cap I_\Gamma$. By setting $vis(\gamma) = \varepsilon$ whenever γ consists entirely of invisible variables, we allow the system to completely hide a push operation (obviously, a corresponding pop will also be invisible). When such a push occurs, the environment does not see the symbol $\emptyset$ being pushed, rather, it sees no push at all. This is necessary since in many applications what is actually pushed is immaterial, and the information to be revealed or hidden is only the depth of the pushdown store. The visible part of a pushdown store content $s = \gamma_0 \cdots \gamma_n \cdot \natural$ is defined in the natural way: $vis(s) = vis(\gamma_0) \cdots vis(\gamma_n) \cdot \natural$. The visible part of a configuration (q, α), is thus $vis((q,\alpha)) = (vis(q), vis(\alpha))$. As for modules, the designation of a configuration of an *OPD* as an environment configuration is known to the environment. Thus, we require that for every two configurations (q, α) and (q', α') such that $vis(q, top(\alpha)) = vis(q', top(\alpha'))$, it holds that $(q, top(\alpha)) \in Env$ iff $(q', top(\alpha')) \in Env$.

Definition 2. *An* OPD $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \natural, \delta, \mu, Env \rangle$ *induces an infinite-state module* $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong \rangle$*, where:*

- *AP is a set of atomic propositions;*
- $W_s \cup W_e = Q \times \Gamma^* \cdot \flat$ *is the set of configurations;*
- W_e *is the set of configurations* (q, α) *such that* $(q, top(\alpha)) \in Env$*;*
- $w_0 = (q_0, \flat)$ *is the initial configuration;*
- R *is a transition relation, where* $((q, \gamma \cdot \alpha), (q', \beta)) \in R$ iff *there exist* $((q, \gamma), (q', \beta')) \in \delta$ *such that* $\beta = \beta' \cdot \alpha$*;*
- $L((q, \alpha)) = \mu(q, top(\alpha))$ *for all* $(q, \alpha) \in W$*;*
- *For every* $w, w' \in W$*, we have that* $w \cong w'$ *iff* $vis(w) = vis(w')$*.*

To describe the interaction of an *OPD* $\mathcal{S}$ with its environment, we consider the interaction of the environment with the induced module $\mathcal{M}_S$. Indeed, every environment ξ of $\mathcal{S}$, can be represented by a strategy tree $\langle [W]^*, \xi \rangle$, and the composition $\mathcal{M}_S \lhd \xi$ of $\langle [W]^*, \xi \rangle$ with $\langle T_{\mathcal{M}_S}, V_{\mathcal{M}_S} \rangle$ describes all the computations of $\mathcal{S}$ allowed by the environment ξ. We can thus define the following problem.

Pushdown module checking problem with imperfect information. Given an *OPD* $\mathcal{S}$, and a temporal logic formula[3] φ, the pushdown module checking problem with imperfect information is to decide whether $\mathcal{M}_S \models_r \varphi$; i.e., to decide whether $\mathcal{M}_S \lhd \xi$ satisfies φ, for every environment ξ for which $\mathcal{M}_S \lhd \xi$ is deadlock free.

Note that starting with an *OPD* $\mathcal{S}$ having $Env = \emptyset$ (that is, the behavior of $\mathcal{S}$ is not affected by any environment) the induced module is closed. In this case, the problem we address becomes the classical *pushdown model checking problem*, and for for branching-time specifications it has been first studied in [37, 38]. Also, if the *OPD* is open ($Env \neq \emptyset$) but there is no invisible information (both H and H_Γ are empty), the addressed problem is called *pushdown module checking with perfect information*, and for *CTL* and *CTL** specifications it has been studied in [10].

[3]The semantics of temporal logics such as *CTL* is usually defined with respect to infinite paths, so we assume $\mathcal{M}_S$ has no configurations without successors. However, using a similar technique to the one used in [10] our results can be adapted to the situation where terminal configurations are also allowed.

4. Undecidability of the General Case

In this section, we study the pushdown module checking problem with imperfect information and show that it is undecidable already for the case of *CTL* specifications. In the next section, we show that undecidability relies on the system's ability to hide information about the pushdown store. Namely, we prove that if we start with an *OPD* with $H_\Gamma = \emptyset$, the problem becomes decidable (even if $H \neq \emptyset$), and its complexity is the same as that of pushdown module checking with perfect information.

Undecidability of the pushdown module checking problem with imperfect information is obtained by a reduction from the universality problem of nondeterministic pushdown automata on finite words (*PDA*), which is undecidable [23]. That is, given a *PDA* $\mathcal{P}$, we build an *OPD* $\mathcal{S}$ and a *CTL* formula φ, such that the module induced by $\mathcal{S}$ reactively satisfies φ iff $\mathcal{P}$ is universal.

Our choice to do a reduction from the universality problem of PDA is not at all arbitrary. It is well known that checking for the universality of a nondeterministic automaton can be thought of as a game between a protagonist trying to prove that the automaton is not universal, and an antagonist which claims that it is universal. The universality game is played as follows. The protagonist chooses the first input letter, the antagonist responds with the first part of the run, the protagonist chooses the next input letter, the antagonist extends the run, and so on. The protagonist wins if the resulting run is rejecting, and the antagonist wins if it is accepting. Note that if the automaton is not universal then the protagonist has a winning strategy: choosing the letters of a word not accepted by the automaton. However, since the automaton is nondeterministic, the converse is not true. That is, even if the automaton is universal, the antagonist may not have a winning strategy. Due to nondeterminism, if the protagonist can see the moves of the antagonist then it may force the run to be rejecting even though the word it supplies can be accepted by the automaton. Hence, the game is sound but not complete. However, if the protagonist cannot see the moves of the antagonist the game becomes sound and complete. Deciding if the automaton is not universal can be reduced to deciding whether the protagonist has a winning strategy in the corresponding universality game with imperfect information. By casting the universality game of *PDA* to a special instance of the pushdown module checking problem with imperfect information, the latter is shown to be undecidable.

Theorem 1. CTL *pushdown module-checking with imperfect information is undecidable.*

Proof. Given a *PDA* $\mathcal{P}$, we build an *OPD* $\mathcal{S}$ and a *CTL* formula φ, such that the module induced by $\mathcal{S}$ reactively satisfies φ iff $\mathcal{P}$ is universal. Let $\mathcal{P} = \langle \Sigma, \Gamma, Q, q_0, \flat, \Delta, F \rangle$ be a *PDA* on finite words, with an input alphabet Σ, a pushdown store alphabet Γ, a set Q of states, an initial state q_0, a bottom of pushdown store symbol $\flat$, a transition function $\Delta : Q \times \Sigma \times \Gamma_\flat \to 2^{Q \times \Gamma_\flat^*}$, and a set of accepting states $F \subseteq Q$. We assume without loss of generality that $\mathcal{P}$ never gets stuck on any input.

The *OPD* $\mathcal{S}$ simulates all the runs of $\mathcal{P}$ on all words in Σ^*. The states of $\mathcal{S}$ are pairs of a state in Q and a letter in Σ. Each transition of $\mathcal{P}$, that reads a letter σ moves to a state q and does some pushdown store operation, is simulated in $\mathcal{S}$ by a transition that goes to the state (q, σ) and does the same pushdown store operation. In order to have in $\mathcal{S}$ infinite computations that simulate runs of $\mathcal{P}$ on finite words, we allow $\mathcal{S}$, at any point, to end the simulation of a run by moving to one of two special states q_{acc} and q_{rej}, depending on whether the computation corresponds to an accepting or a rejecting run of $\mathcal{P}$, respectively. Once in q_{acc} or q_{rej}, the computation stays there forever. The visible part of a configuration $((q, \sigma), \alpha)$ of $\mathcal{S}$ is just σ. Thus, looking at a computation of $\mathcal{S}$ that simulates a run of $\mathcal{P}$ on a word $\sigma_1 \cdots \sigma_n$, the environment can only see the letters $\sigma_1, \ldots, \sigma_n$. It follows that the environment cannot distinguish between computations of $\mathcal{S}$ that correspond to different runs of $\mathcal{P}$ on the same word. This ensures that the environment cannot disable some, but not all, of these computations. Note that a word $w \in \Sigma^*$ is accepted by $\mathcal{P}$ iff there is a computation in $\mathcal{S}$, corresponding to a run of $\mathcal{P}$ on w, that visits the state q_{acc}. The formula φ will check this condition. Formally, let $\mathcal{P} = \langle \Sigma, \Gamma, Q, q_0, \flat, \Delta, F \rangle$ be a *PDA* on finite words. We build an *OPD* $\mathcal{S} = \langle AP, Q', q_0', \Gamma', \flat, \delta, \mu, Env \rangle$ where,

- $AP = \Sigma \cup \{\sharp, Acc\}$, where $\sharp$ and Acc are new symbols not in Σ (nor in Q).
- $I = \Sigma \cup \{\sharp\}$, and $H = Q \cup \{Acc\}$. The set of states Q' is $\{\{q, \sigma\} : q \in Q, \sigma \in \Sigma\} \cup \{\{\sharp\}, \{\sharp, Acc\}\}$. For simplicity, we will identify a set $\{q, \sigma\}$ with the pair (q, σ), and use the aliases $q_{\text{acc}} = \{\sharp, Acc\}$ and $q_{\text{rej}} = \{\sharp\}$.
- $q_0' = (q_0, \sigma_0)$ where σ_0 is a letter arbitrarily chosen from Σ.

- $I_\Gamma = \emptyset$ and $H_\Gamma = \Gamma$. The pushdown store alphabet Γ' is formally the subset $\{\{\gamma\} : \gamma \in \Gamma\}$ of $2^{I_\Gamma \cup H_\Gamma}$. However, we can obviously simplify and set $\Gamma' = \Gamma$.

- δ is defined as follows. For all $(p, \sigma) \in Q \times \Sigma$ and $\gamma \in \Gamma_\flat$, we have that $(((p, \sigma), \gamma), ((q, \sigma'), \beta)) \in \delta$ iff $(q, \beta) \in \Delta(p, \sigma', \gamma)$. Also, $(((p, \sigma), \gamma), (q_{\text{acc}}, \gamma)) \in \delta$ iff $p \in F$,
and $(((p, \sigma), \gamma), (q_{\text{rej}}, \gamma)) \in \delta$ iff $p \notin F$. Finally, $((q, \gamma), (q, \gamma)) \in \delta$ for $q \in \{q_{\text{acc}}, q_{\text{rej}}\}$.

- μ is defined as follows. For every $(q, \sigma) \in Q \times \Sigma$ and $\gamma \in \Gamma_\flat$ we have that, $\mu((q, \sigma), \gamma) = \{\sigma\}$. Also, $\mu(q_{\text{acc}}, \gamma) = \{\sharp, Acc\}$ and $\mu(q_{\text{rej}}, \gamma) = \{\sharp\}$.

- $Env = Q \times \Gamma_\flat$. That is, $\mathcal{S}$ has only environment configurations.

Let $\mathcal{M}_S = \langle AP, \emptyset, W, w_0, R, L, \cong \rangle$ be the module induced by $\mathcal{S}$. Observe that by our choice of visible control and pushdown store variables, the set of equivalence classes $[W]$ of the configurations of $\mathcal{M}_S$ is $\{(\sigma, \flat) : \sigma \in \Sigma \cup \{\sharp\}\}$. We can safely ignore the constant $\flat$ component of each pair, and think of environment strategies as full $\{\top, \bot\}$-labeled $(\Sigma \cup \{\sharp\})$-trees. We claim that $\mathcal{P}$ is universal if and only if $\mathcal{M}_S \models_r \varphi$, where $\varphi = EG\neg\sharp \vee EFAcc$. Recall that the environment represents the protagonist in our scenario, and its aim is to find a word that $\mathcal{P}$ does not accept. Thus, the environment would like to find a strategy ξ such that $\mathcal{M}_S \lhd \xi$ does not satisfy φ. However, if $\mathcal{P}$ is universal, it would not be able to do so. Intuitively, given a strategy ξ, the tree $\mathcal{M}_S \lhd \xi$ satisfies $\varphi_1 = EG\neg\sharp$ if the pruning done by ξ leaves in the tree a path that represents a run of $\mathcal{P}$ on an infinite word (and thus should not be considered a success of the environment in finding a word not accepted by $\mathcal{P}$, since we only care about runs of $\mathcal{P}$ on finite words); on the other hand, if the tree $\mathcal{M}_S \lhd \xi$ satisfies $\varphi_2 = EFAcc$, then it contains an accepting run of $\mathcal{P}$ on some finite word. More technically, the sub-formula φ_1 is satisfied by the tree $\mathcal{M}_S \lhd \xi$ iff ξ has an infinite path $\pi = v_1 \cdot v_2 \cdots$ such that for every $i \geq 0$ we have that v_i is labeled with $\top$, and $last(v_i) \neq \sharp$. The proof follows by showing that for all other strategies ξ, the tree $\mathcal{M}_S \lhd \xi$ satisfies the sub-formula φ_2 iff $\mathcal{P}$ is universal.

Given a word $w = \sigma_1 \cdots \sigma_k \in \Sigma^*$, and a run $r = (q_0, \flat) \cdot (q_1, \alpha_1) \cdots (q_k, \alpha_k)$ of $\mathcal{P}$ on w, let $\tau = (q_0', \flat) \cdot ((q_1, \sigma_1), \alpha_1) \cdots ((q_k, \sigma_k), \alpha_k)$ be the finite computation of $\mathcal{M}_S$ corresponding to r. The visible part of τ is $vis(\tau) =$

$(\sigma_0, \flat) \cdot (\sigma_1, \flat) \cdots (\sigma_k, \flat)$. Thus, given a strategy ξ, we have that τ is associated with the node w in the strategy tree $\langle [W]^*, \xi \rangle$ (recall that $(\sigma_0, \flat)$ is associated with the root ε). It follows that all the nodes in $\langle T_{\mathcal{M}_S}, V_{\mathcal{M}_S} \rangle$ corresponding to runs of $\mathcal{P}$ on w are associated with the same node of the strategy tree. Hence, a strategy can either enable all computations corresponding to runs of $\mathcal{P}$ on w, or disable them all. Note that given a run r of $\mathcal{P}$ on w, with a corresponding finite computation $\tau = (q'_0, \flat) \cdot ((q_1, \sigma_1), \alpha_1) \cdots ((q_k, \sigma_k), \alpha_k)$ of $\mathcal{M}_S$ as above, the configuration $(q_{\mathrm{acc}}, \alpha_k)$ is a successor of $((q_k, \sigma_k), \alpha_k)$ iff r is an accepting run, and $(q_{\mathrm{rej}}, \alpha_k)$ is a successor of $((q_k, \sigma_k), \alpha_k)$ iff r is a rejecting run. Thus, τ can be extended to a path witnessing the satisfaction of φ_2 iff r is an accepting run of $\mathcal{P}$ on w.

For every word $w = \sigma_1 \dots \sigma_k \in \Sigma^*$ there is a special strategy ξ_w that enables exactly the computations in the module corresponding to all of $\mathcal{P}$'s runs on w. The strategy ξ_w has all nodes on the path $\sigma_1 \cdots \sigma_k \cdot \sharp^\omega$ marked with $\top$ and all other nodes marked with $\bot$. It is easy to see that $\mathcal{M}_S \lhd \xi_w$ is deadlock free, that $\mathcal{M}_S \lhd \xi_w \not\models \varphi_1$, and that w is accepted by $\mathcal{P}$ iff $\mathcal{M}_S \lhd \xi_w \models \varphi_2$. Hence, to complete the proof, it is sufficient to show that if $\mathcal{P}$ is universal then for every other strategy ξ, for which $\mathcal{M}_S \lhd \xi$ has a node x whose label contains $\sharp$, we have that $\mathcal{M}_S \lhd \xi \models \varphi_2$. Let x be such a node of minimal depth, and let τ be the father of x. Note that τ must be of the form $\tau = (q'_0, \flat) \cdot ((q_1, \sigma_1), \alpha_1) \cdots ((q_k, \sigma_k), \alpha_k)$. Consider the word $w = \sigma_1 \cdots \sigma_k$. Since ξ cannot distinguish between computations corresponding to different runs of $\mathcal{P}$ on w, the tree $\mathcal{M}_S \lhd \xi$ must contain not only τ, but also the computations corresponding to all other runs (if such runs exist) of $\mathcal{P}$ on w. Thus, if $\mathcal{P}$ is universal, $\mathcal{M}_S \lhd \xi$ contains a path $\pi = \tau' \cdot (q_{\mathrm{acc}}, \alpha'_k)^\omega$, where $\tau' = (q'_0, \flat) \cdot ((q'_1, \sigma'_1), \alpha'_1) \cdots ((q'_k, \sigma'_k), \alpha'_k)$ is a finite computation (maybe τ) corresponding to an accepting run of $\mathcal{P}$ on w. Since the configuration $(q_{\mathrm{acc}}, \alpha'_k)$ is labeled with $\{\sharp, Acc\}$, the path π is a witness for the satisfaction of φ_2. ■

It is easy to see that we can replace the *OPD* $\mathcal{S}$ used in the proof of Theorem 1 by an *OPD* S' with only one state. $\mathcal{S}'$ uses as pushdown store alphabet pairs of a control state and a pushdown store symbol of $\mathcal{S}$, and can thus remember the current control state of $\mathcal{S}$ in its (invisible) top of pushdown store. This implies the following corollary to Theorem 1:

Corollary 1. *The pushdown module checking problem with imperfect information is undecidable also when the control states are completely visible.*

A somewhat more surprising result is that the pushdown module checking

problem remains undecidable even if the environment has full information, not only about the control states, but also about which atomic propositions hold at each and every configuration of the system.

Theorem 2. *The imperfect information pushdown module checking problem for* CTL*, with visible control states and atomic propositions, is undecidable.*

Proof. Observe that almost all the atomic propositions of the *OPD* $\mathcal{S}$ used in the proof of Theorem 1 are visible. The only violation is that for every $\alpha, \alpha' \in \Gamma^* \cdot \flat$, we have that $(q_{\text{acc}}, \alpha) \cong (q_{\text{rej}}, \alpha')$, but $\{\sharp, Acc\} = L(q_{\text{acc}}, \alpha) \neq L(q_{\text{rej}}, \alpha') = \{\sharp\}$. Since the formula used in the proof is $\varphi = EG\neg\sharp \vee EFAcc$, keeping the environment in the dark as to whether only $\sharp$ holds, or both $\sharp$ and *Acc* hold, is crucial. Indeed, if we fully expose the atomic propositions, we would make legal the environment ξ that prunes only the computations corresponding to accepting runs of $\mathcal{P}$, with the consequence that $\mathcal{M}_S \lhd \xi \not\models \varphi$ even in cases where $\mathcal{P}$ is universal. However, the environment's ability to prune is not only limited by the information visible to it, but also by the requirement that it does not completely block the system. With a slight modification to the construction of the *OPD* $\mathcal{S}$ used in the proof of Theorem 1, we can have visible atomic propositions, but reveal the difference between computations corresponding to accepting and rejecting runs only when it is too late for the environment to prune based on that difference. This is done by changing $\mathcal{S}$ in such a way that a simulation $\tau = (q'_0, \flat) \cdot ((q_1, \sigma_1), \alpha_1) \cdots ((q_k, \sigma_k), \alpha_k)$ of an accepting run r of $\mathcal{P}$ is not ended by moving directly to the sink configuration $(q_{\text{acc}}, \alpha_k)$. Instead, we temporarily move to the configuration $(q_{\text{rej}}, \{\surd\} \cdot \alpha_k)$. The only possible move from $(q_{\text{rej}}, \{\surd\} \cdot \alpha_k)$ is to the configuration $(q_{\text{acc}}, \alpha_k)$.

Formally, we make the following modifications to $\mathcal{S}$. We make the control variable *Acc* visible by setting $I = \Sigma \cup \{\sharp, Acc\}$, and $H = Q$. We add a new invisible pushdown store variable $\surd$, and derive from it a new pushdown store symbol $\{\surd\}$. The definition of the labeling function μ remains the same, except that it now ranges over the extended pushdown store alphabet. Finally, we replace every transition of the form $(((p, \sigma), \gamma), (q_{\text{acc}}, \gamma))$ with the transitions $(((p, \sigma), \gamma), (q_{\text{rej}}, \{\surd\} \cdot \gamma))$ and $((q_{\text{rej}}, \{\surd\}), (q_{\text{acc}}, \varepsilon))$. Let τ and τ' be computations of $\mathcal{S}$ corresponding to a rejecting run and an accepting run (respectively) of $\mathcal{P}$ on the same word w. The key observation is that the first point of difference an environment ξ sees between the path $\pi = \tau \cdot (q_{\text{rej}}, \alpha)^\omega$ and the path $\pi' = \tau' \cdot (q_{\text{rej}}, \{\surd\} \cdot \alpha') \cdot (q_{\text{acc}}, \alpha')^\omega$, is at the nodes

$v = \tau \cdot (q_{\text{rej}}, \alpha) \cdot (q_{\text{rej}}, \alpha)$ and $v' = \tau' \cdot (q_{\text{rej}}, \{\surd\} \cdot \alpha') \cdot (q_{\text{acc}}, \alpha')$. But by now, it is too late for the environment to prune without creating a deadlock in $\mathcal{M}_S \lhd \xi$. This is because v is the only successor of its father, and so is v'. Combining the above with Corollary 1 completes the proof. ■

Observe that the formula used in the proof of Theorems 1 and 2 is an existential formula. Hence, the problem is already undecidable for the existential fragment *ECTL* of *CTL*. Obviously, the problem remains undecidable for more expressive logics such as CTL^* and μ-calculus.

5. Semi-Alternating Pushdown Tree Automata

In this section, we introduce semi-alternating pushdown tree automata, and prove their equivalence to nondeterministic Büchi tree automata. The results of this section are used in subsequent sections to solve the pushdown module checking problem with imperfect state information and visible pushdown store.

Alternating pushdown tree automata [31, 25], are alternating tree automata, augmented with a pushdown store. Semi-alternating pushdown tree automata are obtained by restricting the universality with respect to the pushdown store. The formal definition of semi-alternating pushdown tree automata follows[4].

A *semi-alternating pushdown tree automaton* is a tuple $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F \rangle$ where Σ is a finite input alphabet, D is a finite set of *directions*, Γ is a finite pushdown store alphabet, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\flat \notin \Gamma$ is the pushdown store bottom symbol, and F is an acceptance condition, to be defined later. δ is a finite transition function $\delta : Q \times \Sigma \times \Gamma_\flat \to \mathcal{B}^+(D \times Q \times \Gamma_\flat^*)$, where $\Gamma_\flat = \Gamma \cup \{\flat\}$ as usual, and $\mathcal{B}^+(D \times Q \times \Gamma_\flat^*)$ is the set of all positive boolean combinations of triples (d, q, β), where d is a direction, q is a state, and β is a string of pushdown store symbols. We also allow the formulas **true** and **false**. We write $S \in \delta(p, \sigma, \gamma)$ to denote that S is a set of tuples (d, q, β) that satisfy $\delta(p, \sigma, \gamma)$. What makes the automaton semi-alternating is the requirement that for every $d \in D$, $\sigma \in \Sigma$, $p, p' \in Q$ (possibly the same state), and $\gamma \in \Gamma$, if (d, q, β) appears in $\delta(p, \sigma, \gamma)$, and (d, q', β') appears in $\delta(p', \sigma, \gamma)$, then $\beta = \beta'$. That is, two copies of the automaton that read the same input, from two configurations

[4]Note that our semi-alternating tree automata should not be confused with Ibara's semi-alternating stack automata [20]

that have the same top symbol of the pushdown store and proceed in the same direction, must push the same value into the pushdown store. In particular, it follows that in every run, two copies of the automaton that are reading the same node of an input tree have the same pushdown store content. Note that if we remove the semi-alternation requirement, the resulting automaton is simply an *alternating pushdown tree automaton*.

For example, having $\delta(q, \sigma, \gamma) = ((0, q_1, \beta_1) \vee (1, q_2, \beta_2)) \wedge (1, q_1, \beta_2)$ means that when a copy of the automaton that is in a configuration where the current state is q, and the top of pushdown store is γ, reads a node in the input tree whose label is σ, it can proceed in one of two ways. In the first option, one copy proceeds in direction 0 to state q_1, by replacing γ with β_1, and one copy proceeds in direction 1 to state q_1, by replacing γ with β_2. In the second option, two copies proceed in direction 1, one to state q_1 and the other to state q_2, and in both copies γ is replaced with β_2. Hence, $\vee$ and $\wedge$ in $\delta(q, \sigma, \gamma)$ represent, respectively, choice and concurrency. As a special case of semi-alternating pushdown tree automata, we consider *nondeterministic pushdown tree automata* where the concurrency feature is not allowed. That is, whenever the automaton visits a node x of the input tree, it sends to each successor (direction) of x at most one copy of itself. More formally, a nondeterministic pushdown tree automaton is a semi-alternating pushdown tree automaton in which δ is in disjunctive normal form, and in each conjunct each direction appears at most once.

A run of a semi-alternating pushdown tree automaton $\mathcal{A}$ on a Σ-labeled tree $\langle T, V \rangle$, with $T = D^*$, is a $(D^* \times Q \times \Gamma^* \cdot \flat)$-labeled $\mathbb{N}$-tree $\langle T_r, r \rangle$ such that the root is labeled with $(\varepsilon, q_0, \flat)$ and the labels of each node and its successors satisfy the transition relation. Formally, a $(D^* \times Q \times \Gamma^* \cdot \flat)$-labeled tree $\langle T_r, r \rangle$ is a run of $\mathcal{A}$ on $\langle T, V \rangle$ iff

- $r(\varepsilon) = (\varepsilon, q_0, \flat)$, and
- for all $x \in T_r$ such that $r(x) = (y, p, \gamma \cdot \alpha)$, there is an $n \in \mathbb{N}$ such that the successors of x are exactly $x \cdot 1, \ldots x \cdot n$, and for all $1 \leq i \leq n$ we have $r(x \cdot i) = (y \cdot d_i, p_i, \beta_i \cdot \alpha)$ for some $\{(d_1, p_1, \beta_1), \ldots, (d_n, p_n, \beta_n)\} \in \delta(p, V(y), \gamma)$.

As for tree automata without a pushdown store, a run $\langle T_r, r \rangle$ is accepting iff all its infinite paths satisfy the acceptance condition. Note that here we are only interested in the Büchi and parity acceptance conditions. A parity

winning condition F maps all states of the automaton to a finite set of colors $C = \{C_{\min}, \ldots, C_{\max}\} \subset \mathbb{N}$. Thus, $F : Q \to C$. For a path π, let $maxC(\pi)$ be the maximal color that appears infinitely often along π. Then, π satisfies the parity condition F iff $maxC(\pi)$ is even. The Büchi acceptance condition is a special case of the parity condition with only two colors, i.e., $C = \{1, 2\}$.

We denote the different classes of pushdown tree automata by preceding with "PD-" a three letter acronyms in $\{\mathrm{A}, \mathrm{S}, \mathrm{N}\} \times \{\mathrm{B}, \mathrm{P}\} \times \{\mathrm{T}\}$, where the first letter stands for the branching mode of the automaton (alternating, semi-alternating, or nondeterministic); the second letter stands for the acceptance-condition type (Büchi, or parity); and the third letter indicates that the automaton runs on trees. Thus, for example, a semi-alternating pushdown tree automaton with a parity acceptance condition is a PD-SPT, and a nondeterministic pushdown tree automaton with a Büchi acceptance condition is a PD-NBT.

Given a pushdown tree automaton $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F \rangle$, we define the size of $\mathcal{A}$ as $|\mathcal{A}| = |Q| + |\delta|$, where $|\delta|$ is the sum of the lengths of the satisfiable (i.e., not **false**) formulas that appear in $\delta(q, \sigma, \gamma)$ for some q, σ, and γ.

5.1. Translating PD-SPT to PD-NPT

As mentioned in Section 1, alternating pushdown automata are not equivalent to nondeterministic ones. However, as we show here, the limitations imposed on semi-alternating automata allow us to translate a PD-SPT to an equivalent PD-NPT. A key observation is that since a pushdown store operation performed by a semi-alternating automaton does not depend on the current (or next) control states, we can split the transition function of a PD-SPT into two functions: a *state transition function* δ_Q, and a *pushdown store update function* δ_Γ, as follows. Given a PD-SPT $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F \rangle$, let $\delta_Q : Q \times \Sigma \times \Gamma_\flat \to \mathcal{B}^+(D \times Q)$ be the projection of δ on $\mathcal{B}^+(D \times Q)$. That is, $\delta_Q(q, \sigma, \gamma)$ is obtained from $\delta(q, \sigma, \gamma)$ by replacing every element (d, q, β) that appears in $\delta(q, \sigma, \gamma)$ with (d, q). The pushdown store update function $\delta_\Gamma : \Sigma \times \Gamma_\flat \times D \to \Gamma_\flat^*$, is a partial function; for every $(p, \sigma, \gamma) \in Q \times \Sigma \times \Gamma_\flat$ and every $(d, q, \beta) \in D \times Q \times \Gamma_\flat^*$, such that (d, q, β) appears in $\delta(p, \sigma, \gamma)$, we let $\delta_\Gamma(\sigma, \gamma, d) = \beta$. Since $\mathcal{A}$ is semi-alternating, δ_Γ is well defined. Observe that for every $(p, \sigma, \gamma) \in Q \times \Sigma \times \Gamma_\flat$ we have that $\delta(p, \sigma, \gamma)$ can be obtained from $\delta_Q(p, \sigma, \gamma)$ by replacing every (d, q) that appears in $\delta_Q(p, \sigma, \gamma)$ with $(d, q, \delta_\Gamma(\sigma, \gamma, d))$.

Consider a Σ-labeled tree $\langle T, V\rangle$, with $T = D^*$. Note that for every node $x \in T$ and every run of $\mathcal{A}$ on $\langle T, V\rangle$, the pushdown store content of all the copies of $\mathcal{A}$ that visit x is the same, and only depends on x. We can thus define a function $\Delta_\Gamma : T \to \Gamma_\flat^*$, giving for every node x its associated pushdown store content, as follows: (1) $\Delta_\Gamma(\varepsilon) = \flat$, and (2) for all $x \cdot d \in T$ we have $\Delta_\Gamma(x \cdot d) = \delta_\Gamma(V(x), \gamma, d) \cdot \beta$, where $\Delta_\Gamma(x) = \gamma \cdot \beta$, and $\gamma \in \Gamma_\flat$.

Annotating input trees with pushdown store symbols enables us to simulate a PD-SPT by an APT running on the annotated version of an input tree. Given a Σ-labeled tree $\langle T, V\rangle$, we define its $\Gamma_\mathcal{A}$-*annotation* to be the $(\Sigma \times \Gamma_\flat)$-labeled tree $\langle T, U\rangle$, obtained by letting $U(x) = (V(x), \mathit{top}(\Delta_\Gamma(x)))$, for every $x \in T$.

Lemma 1. *Let $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F\rangle$ be a PD-SPT. There is an APT $\tilde{\mathcal{A}}$, such that $\mathcal{A}$ accepts $\langle T, V\rangle$ iff $\tilde{\mathcal{A}}$ accepts the $\Gamma_\mathcal{A}$-annotation of $\langle T, V\rangle$.*

Proof. Consider the APT $\tilde{\mathcal{A}} = \langle \Sigma \times \Gamma_\flat, D, Q, q_0, \tilde{\delta}, F\rangle$, where $\tilde{\delta}(q, (\sigma, \gamma)) = \delta_Q(q, \sigma, \gamma)$. It is not hard to see that every run $r = \langle T_r, r\rangle$ of $\mathcal{A}$ on $\langle T, V\rangle$ induces a corresponding run $r' = \langle T_r, r'\rangle$ of $\tilde{\mathcal{A}}$ on the $\Gamma_\mathcal{A}$-annotation of $\langle T, V\rangle$, and vice versa. The connection between r and r' being that for every $x \in T_r$, we have that $r(x) = (y, p, \alpha)$ iff $r'(x) = (y, p)$ and $\Delta_\Gamma(x) = \alpha$. ■

By [33], every APT can be translated to an equivalent NPT. Hence, Lemma 1 implies that if $\mathcal{A}$ is a PD-SPT, then there is an NPT $\mathcal{A}'$ such that $\mathcal{A}$ accepts $\langle T, V\rangle$ iff $\mathcal{A}'$ accepts the $\Gamma_\mathcal{A}$-annotation of $\langle T, V\rangle$. This allows us to translate $\mathcal{A}$ to an equivalent PD-NPT $\mathcal{A}''$ (running on the same input trees as $\mathcal{A}$). Given a Σ-labeled tree, $\mathcal{A}''$ generates on the fly its $\Gamma_\mathcal{A}$-annotation and runs A' on the annotated tree. Formally, we have the following:

Theorem 3. *A PD-SPT $\mathcal{A}$ with n states and index k can be translated to an equivalent PD-NPT with $(nk)^{O(nk)}$ states, an $O(nk)$ index, and a transition relation of size $(nk)^{O(|D|nk)}$.*

Proof. Let $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F\rangle$ be a PD-SPT and $\tilde{\mathcal{A}} = \langle \Sigma \times \Gamma_\flat, D, Q, q_0, \tilde{\delta}, F\rangle$ be an APT derived from $\mathcal{A}$ by Lemma 1. By [33], $\tilde{\mathcal{A}}$ has an equivalent NPT $\mathcal{A}' = \langle \Sigma \times \Gamma_\flat, D, Q', q_0', \delta', F'\rangle$. Consider the PD-NPT $A'' = \langle \Sigma, D, \Gamma, Q', q_0', \flat, \delta'', F'\rangle$, where for every $(p, \sigma, \gamma) \in Q' \times \Sigma \times \Gamma_\flat$, we have that $\delta''(p, \sigma, \gamma)$ is obtained from $\delta'(p, (\sigma, \gamma))$ by replacing every (d, q) that appears in $\delta'(p, (\sigma, \gamma))$, with $(d, q, \delta_\Gamma(\sigma, \gamma, d))$. Since $\mathcal{A}'$ is nondeterministic, so is $\mathcal{A}''$. Given a Σ-labeled tree $\langle T, V\rangle$, it is not hard to see that for every $x \in T$,

the pushdown store of every copy of $\mathcal{A}''$ that visits x contains exactly $\Delta_\Gamma(x)$. Hence, $\mathcal{A}''$ accepts $\langle T, V\rangle$ iff $\mathcal{A}'$ accepts the $\Gamma_\mathcal{A}$-annotation of $\langle T, V\rangle$, thus, by Lemma 1, iff $\mathcal{A}$ accepts $\langle T, V\rangle$.

We now analyze the blow-up involved in the above construction. Looking at the automata transformations involved, we see that the only transformation that incurs a blow-up in the size of the automaton is the transformation of the APT $\tilde{\mathcal{A}}$ to the NPT $\mathcal{A}'$. By [33], if the APT $\tilde{\mathcal{A}}$ has n states and index k, and it runs over D^* trees, the resulting NPT $\mathcal{A}'$ has $(nk)^{O(nk)}$ states, an $O(nk)$ index, and a transition relation of size $(nk)^{O(|D|nk)}$. We note that the runtime of the algorithm in [33] is polynomial in the size of its input and output automata. We also wish to draw the reader's attention to the fact that the blow-up in the number of states of this translation is independent of the size of the transition relation of $\mathcal{A}$. ■

By [25], the emptiness of a PD-NPT can be decided in time exponential in the product of the number of states, the index, and the size of its transition relation. Together with Theorem 3, this gives us the following corollary:

Corollary 2. *The emptiness problem for a PD-SPT with n states and index k, running on D^* trees, can be solved in time double-exponential in $|D|nk$.*

Since the Büchi acceptance condition can be thought of as a parity condition with only two colors, Theorem 3 also yields a translation from PD-SBT to PD-NPT, and thus we also have the following corollary:

Corollary 3. *The emptiness problem for a PD-SBT with n states, running on D^* trees, can be solved in time double-exponential in $|D|n$.*

5.2. Translating PD-SBT to PD-NBT

As noted above, Theorem 3 also yields a translation from PD-SBT to PD-NPT, which, as it turns out, is good enough to obtain the required complexity results for our intended application in the context of *CTL* pushdown module-checking. However, an alternative route is to use the much simpler and more direct translation of PD-SBT to PD-NBT presented below. In [32] Miyano and Hayashi describe a translation of alternating Büchi automata on words to nondeterministic ones. In [28] the construction is adapted to the translation of alternating Büchi automata on trees to nondeterministic ones. Here, we further extend it to obtain a translation of PD-SBT to PD-NBT.

Theorem 4. *Let $\mathcal{A}$ be a PD-SBT with n states. There is a PD-NBT $\mathcal{A}'$ with $2^{O(n)}$ states, such that $L(\mathcal{A}') = L(\mathcal{A})$.*

Proof. The automaton $\mathcal{A}'$ guesses a subset construction applied to a run of $\mathcal{A}$. At a given node x of a run of $\mathcal{A}'$, it keeps in its memory the set of configurations in which the various copies of $\mathcal{A}$ visit node x in the guessed run. Since $\mathcal{A}$ is semi-alternating, all copies of $\mathcal{A}$ that visit the same node x have the same pushdown store content, and thus, can all be remembered using one pushdown store and a set of states of $\mathcal{A}$. In order to make sure that every infinite path visits states in F infinitely often, $\mathcal{A}'$ keeps track of states that "owe" a visit to F. The details of the construction are given below. Once we establish that indeed one pushdown store is enough to remember the pushdown store of all the copies of $\mathcal{A}$ that visit the same node of the input tree, the correctness of the construction follows from the same arguments used in [32, 28].

Let $\mathcal{A} = \langle \Sigma, D, \Gamma, Q, q_0, \flat, \delta, F \rangle$. Then $\mathcal{A}' = \langle \Sigma, D, \Gamma, 2^Q \times 2^Q, \langle \{q_0\}, \emptyset \rangle, \flat, \delta', 2^Q \times \{\emptyset\} \rangle$. To define δ', we first need the following notation. For a set $S \subseteq Q$, a letter $\sigma \in \Sigma$, and a top of pushdown store symbol $\gamma \in \Gamma$, let $sat(S, \sigma, \gamma)$ be the set of subsets of $D \times Q \times \Gamma_\flat^*$ that satisfy $\bigwedge_{q \in S} \delta(q, \sigma, \gamma)$. Also, for two sets $O \subseteq S \subseteq Q$, a letter $\sigma \in \Sigma$, and a top of pushdown store symbol $\gamma \in \Gamma$, let $pair\_sat(S, O, \sigma, \gamma)$ be such that $\langle S', O' \rangle \in pair\_sat(S, O, \sigma, \gamma)$ iff $S' \in sat(S, \sigma, \gamma)$, $O' \subseteq S'$, and $O' \in sat(O, \sigma, \gamma)$. Finally, for a direction $d \in D$, we have $S'_d = \{s : (d, s, \beta) \in S' \text{ for some } \beta\}$ and $O'_d = \{o : (d, o, \beta) \in O' \text{ for some } \beta\}$. Thus, S'_d and O'_d are, respectively, the collections of all states that appear in S' and O' along with the direction d. Since $\mathcal{A}$ is semi-alternating, for every two triplets (d, q, β) and (d, q', β') in $sat(S, \sigma, \gamma)$ having the same direction d, we have that $\beta = \beta'$. Thus, we can define $store(d, \sigma, \gamma) = \beta$.

Now, δ' is defined, for all $\langle S, O \rangle \in 2^Q \times 2^Q$, $\sigma \in \Sigma$, and $\gamma \in \Gamma$, as follows.

- if $O \neq \emptyset$, then

$$\delta'(\langle S, O \rangle, \sigma, \gamma) = \bigvee_{\substack{\langle S', O' \rangle \in \\ pair\_sat(S, O, \sigma, \gamma)}} \bigwedge_{d \in D} (d, \langle S'_d, O'_d \backslash F \rangle, store(d, \sigma, \gamma))$$

  Thus, when reading σ, from a configuration with a top of pushdown store symbol γ, the automaton $\mathcal{A}'$ sends to a direction $d \in D$ the set S'_d

of states that the different copies of $\mathcal{A}$ send to direction d in the guessed run. Each such S'_d is paired with a subset O'_d of S'_d of the states that still "owe" a visit to F. The key observation is that since $\mathcal{A}$ is semi-alternating, all the copies that $\mathcal{A}$ sends to direction d replace γ with exactly the same pushdown store string, namely, with $store(d, \sigma, \gamma)$. Hence, the pushdown stores of all the copies that $\mathcal{A}$ sends to direction d are identical, and $\mathcal{A}'$ can keep track of them all using the single stack of the copy it sent to direction d.

- if $O = \emptyset$, then

$$\delta'(\langle S, O\rangle, \sigma, \gamma) = \bigvee_{\substack{\langle S', O'\rangle \in \\ pair\_sat(S, O, \sigma, \gamma)}} \bigwedge_{d \in D} (d, \langle S'_d, S'_d \backslash F\rangle, store(d, \sigma, \gamma))$$

Thus, when no state "owes" a visit to F we know that every path in the guessed run of $\mathcal{A}$ visited F one more time, and the requirement to visit F is reinforced.

So, we are done with the proof. ■

6. Module Checking with Visible Pushdown Store

In this section, we show that pushdown module checking with full information about the pushdown store content ($H_\Gamma = \emptyset$), but not about the control states (when $H \neq \emptyset$), is decidable and 2EXPTIME-complete for CTL and propositional μ-calculus specifications, and is 3EXPTIME-complete for CTL^* specifications.

The upper bounds follow by reducing this variant of the pushdown module checking problem to the emptiness problem of PD-SBT for CTL specifications, and to the emptiness problem of PD-SPT for CTL^* and propositional μ-calculus specifications. The lower bounds follow from known results about perfect information pushdown module checking. In Section 6.1 we consider the simpler case of CTL specifications, while the other logics are treated in Section 6.2.

6.1. CTL *Module Checking with Visible Pushdown Store*

Theorem 5. *For an OPD $\mathcal{S}$ with $H_\Gamma = \emptyset$, and a CTL formula φ over $\mathcal{S}$'s atomic propositions, there is a PD-SBT $\mathcal{A}_{\mathcal{S},\varphi}$ of size $O(|\mathcal{S}| \cdot |\varphi|)$ such that $L(A_{\mathcal{S},\varphi})$ is exactly the set of strategies ξ for which $\mathcal{M}_S \lhd \xi$ is deadlock free and satisfies φ.*

Proof. Essentially, the automaton $\mathcal{A}_{\mathcal{S},\varphi}$ we build is an extension of the product automaton obtained in the alternating-automata theoretic approach for *CTL* module checking with imperfect information [26]. The extension we consider concerns the simulation of the pushdown store of the *OPD*, and its correctness follows using the same reasoning found in [26, 29].

Let $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \mu, Env\rangle$ be an OPD, let φ be a *CTL* formula in positive normal form, and let $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong\rangle$ be the module induced by $\mathcal{S}$. We build an automaton $\mathcal{A}_{\mathcal{S},\varphi}$ that accepts $\{\top, \bot\}$-labeled trees corresponding to strategies ξ, whose composition with $\mathcal{M}_S$ is deadlock free and satisfy φ. Intuitively, a run of $\mathcal{A}_{\mathcal{S},\varphi}$ on an input strategy tree ξ proceeds by simulating an unwinding of the module $\mathcal{M}_S$, pruned at each step according to the strategy ξ. Copies of the automaton simulating nodes in the computation tree of $\mathcal{M}_S$ that are indistinguishable by the environment are sent to the same direction in the input tree. The resulting run tree of $\mathcal{A}_{\mathcal{S},\varphi}$ on ξ is basically a replica of the composition $\mathcal{M}_S \lhd \xi$, and the fact that it satisfies the formula φ is checked on the fly, by employing in $\mathcal{A}_{\mathcal{S},\varphi}$ the usual alternating-automata approach for *CTL* model checking. In the full computation tree of $\mathcal{M}_S$, the set of directions is $G = \{(q,\beta) : ((p,\gamma),(q,\beta)) \in \delta$ for some p, q, γ and $\beta\}$. Since in $\mathcal{S}$ the pushdown store is completely visible to the environment, the set of directions of the input strategy trees is $D = \{(vis(q),\beta) : ((p,\gamma),(q,\beta)) \in \delta$ for some p, q, γ and $\beta\}$. Finally, due to the fact that all copies of the automaton sent to direction $(vis(q),\beta)$ push β into the pushdown store, the resulting automaton $\mathcal{A}_{\mathcal{S},\varphi}$ is semi-alternating. Before we give the formal definition of $\mathcal{A}_{\mathcal{S},\varphi}$ we need the following: for $(p, \gamma\cdot\alpha) \in W$, we define the set of successors of $(p,\gamma\cdot\alpha)$ in $\mathcal{M}_S$, to be $s(p,\gamma) = \{(q,\beta) : ((p,\gamma),(q,\beta)) \in \delta\}$. We now formally define $\mathcal{A}_{\mathcal{S},\varphi} = \langle\{\top,\bot\}, D, \Gamma, Q', q_0', \flat, \delta', F\rangle$, where

- $Q' = (Q \times (cl(\varphi) \cup \{p_\top\}) \times \{\forall,\exists\} \times \{p_e, p_s\}) \cup \{q_0'\}$.
- $F = Q \times (\widetilde{U}(\varphi) \cup \{p_\top\}) \times \{\exists,\forall\} \times \{p_e, p_s\}$, where $\widetilde{U}(\varphi)$ is the set of all formulas of the form $\forall\psi_1\widetilde{U}\psi_2$ or $\exists\psi_1\widetilde{U}\psi_2$ in $cl(\varphi)$.

- $\delta' : Q' \times \Sigma \times \Gamma_\flat \to \mathcal{B}^+(D \times Q' \times \Gamma_\flat^*)$ is defined as follows:

  In the rules below, for the sake of succinctness, we consider $m \in \{\exists, \forall\} \times \{p_e, p_s\}$, $h \in AP \cup \{\mathbf{true}, \mathbf{false}\}$. Also, given a transition from $(\langle p, \psi, m\rangle, \top, \gamma)$, we let $p_x = p_e$ if $(p, \gamma) \in Env$, and $p_x = p_s$ otherwise.

  For all $p \in Q$, $\psi_1, \psi_2 \in cl(\varphi)$, $\psi \in cl(\varphi) \cup \{p_\top\}$, and $\gamma \in \Gamma_\flat$, we have:

  - $\delta'(q_0', \bot, \flat) = \mathbf{false}$.
  - $\delta'(q_0', \top, \flat) = \delta'(\langle q_0, p_\top, \exists, p_s\rangle, \top, \flat) \wedge \delta'(\langle q_0, \varphi, \exists, p_s\rangle, \top, \flat)$.
  - $\delta'(\langle p, \psi, \forall, p_e\rangle, \bot, \gamma) = \mathbf{true}$, and $\delta'(\langle p, \psi, \exists, p_e\rangle, \bot, \gamma) = \mathbf{false}$.
  - $\delta'(\langle p, \psi, \forall, p_s\rangle, \bot, \gamma) = \delta'(\langle p, \psi, \forall, p_s\rangle, \top, \gamma)$, and
    $\delta'(\langle p, \psi, \exists, p_s\rangle, \bot, \gamma) = \delta'(\langle p, \psi, \exists, p_s\rangle, \top, \gamma)$.
  - $\delta'(\langle p, p_\top, m\rangle, \top, \gamma) = \bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, p_\top, \exists, p_x\rangle, \beta))$.
  - $\delta'(\langle p, h, m\rangle, \top, \gamma) = \mathbf{true}$ if $h \in \mu((p, \gamma))$, or $h = \mathbf{true}$.
  - $\delta'(\langle p, h, m\rangle, \top, \gamma) = \mathbf{false}$ if $h \notin \mu((p, \gamma))$, or $h = \mathbf{false}$.
  - $\delta'(\langle p, \neg h, m\rangle, \top, \gamma) = \mathbf{true}$ if $h \notin \mu((p, \gamma))$, or $h = \mathbf{false}$.
  - $\delta'(\langle p, \neg h, m\rangle, \top, \gamma) = \mathbf{false}$ if $h \in \mu((p, \gamma))$, or $h = \mathbf{true}$.
  - $\delta'(\langle p, \psi_1 \wedge \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \wedge \delta'(\langle p, \psi_2, m\rangle, \top, \gamma)$.
  - $\delta'(\langle p, \psi_1 \vee \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \vee \delta'(\langle p, \psi_2, m\rangle, \top, \gamma)$.
  - $\delta'(\langle p, \forall X\psi_1, m\rangle, \top, \gamma) = (\bigwedge_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \psi_1, \forall, p_x\rangle, \beta))$.
  - $\delta'(\langle p, \exists X\psi_1, m\rangle, \top, \gamma) = (\bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \psi_1, \exists, p_x\rangle, \beta))$.
  - $\delta'(\langle p, \forall \psi_1 U \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_2, m\rangle, \top, \gamma) \vee$
    $(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \wedge \bigwedge_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \forall \psi_1 U \psi_2, \forall, p_x\rangle, \beta))$.
  - $\delta'(\langle p, \exists \psi_1 U \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_2, m\rangle, \top, \gamma) \vee$
    $(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \wedge \bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \exists \psi_1 U \psi_2, \exists, p_x\rangle, \beta))$.
  - $\delta'(\langle p, \forall \psi_1 \widetilde{U} \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_2, m\rangle, \top, \gamma) \wedge$
    $(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \vee \bigwedge_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \forall \psi_1 \widetilde{U} \psi_2, \forall, p_x\rangle, \beta))$.
  - $\delta'(\langle p, \exists \psi_1 \widetilde{U} \psi_2, m\rangle, \top, \gamma) = \delta'(\langle p, \psi_2, m\rangle, \top, \gamma) \wedge$
    $(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \vee \bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \exists \psi_1 \widetilde{U} \psi_2, \exists, p_x\rangle, \beta))$.

States with the component $p_\top$ are used to check that the composition of $\mathcal{M}_S$ with the strategy is deadlock free, while states with a component in $cl(\varphi)$ check that this composition satisfies φ. The components p_e and p_s are used to flag that a currently simulated node, of the computation tree of $\mathcal{M}_S$, is a child of an environment or a system node, respectively. Clearly, the simulation should respect the strategy pruning specifications only if they correspond to children of environment nodes; that is, only if the current state q contains p_e. Every state is either in an existential or a universal mode, as specified by the $\forall$ and $\exists$ components. When the automaton is in a universal state $(q, \varphi, \forall, p_e)$ with a pushdown store content α, it accepts all strategies for which (q, α) in $\mathcal{M}_S$ is either pruned or satisfies φ (where $p_\top$ is satisfied iff the root of the strategy is labeled $\top$). When the automaton is in an existential state $(q, \varphi, \exists, p_e)$ with a pushdown store content α, it accepts all strategies for which (q, α) in $\mathcal{M}_S$ is not pruned and satisfies φ.

To get a feeling of the transition rules, consider, for example, a transition from the configuration $(\langle p, AX\psi, \exists, p_e\rangle, \gamma \cdot \alpha)$, where $(p, \gamma) \in Env$. First, if the transition to $(p, \gamma \cdot \alpha)$ is disabled (that is, the automaton reads $\bot$), then, as the current mode is existential, the run is rejecting. If the transition to $(p, \gamma \cdot \alpha)$ is enabled, then the successors of $(p, \gamma \cdot \alpha)$ that are enabled should satisfy ψ. Note that all the successors of $(p, \gamma \cdot \alpha)$ that are indistinguishable by the environment are sent by the automaton to the same direction v. This guarantees that either all these successors are enabled by the strategy (in case the letter to be read in direction v is $\top$) or all are disabled (in case the letter in direction v is $\bot$). In addition, since the requirement to satisfy ψ concerns only successors of $(p, \gamma \cdot \alpha)$ that are enabled, the mode of the new states is universal. The copies of $\mathcal{A}_{\mathcal{S},\varphi}$ that check the composition with the strategy to be deadlock free guarantee that at least one successor of $(p, \gamma \cdot \alpha)$ is enabled. As noted earlier, the enable/disable instructions of the strategy are ignored in every configuration $(p, \gamma \cdot \alpha)$ that is a successor of a system configuration. Also note that since we assume that no configuration in $\mathcal{M}_S$ has no successors, the conjunctions and disjunctions in δ' cannot be empty.

It is easy to see that $\mathcal{A}_{\mathcal{S},\varphi}$ has $O(|\mathcal{S}| \cdot |\varphi|)$ states, and it is left to show that $\mathcal{A}_{\mathcal{S},\varphi}$ is semi-alternating. It is sufficient to show that for every $(t, \beta) \in D$, $\sigma \in \Sigma$, $p, p' \in Q'$, and $\gamma \in \Gamma$, if $((t, \beta), p', \beta')$ appears in $\delta'(p, \sigma, \gamma)$ then $\beta = \beta'$. To see that, notice that $((t, \beta), p', \beta')$ appears in $\delta'(p, \sigma, \gamma)$ only if $(q, \beta') \in s(p, \gamma, (t, \beta))$, for some $q \in Q$. By the definition of $s(p, \gamma, (t, \beta))$ we must have that $vis(q, \beta') = (t, \beta)$. Since the pushdown store is completely visible, we have that $vis(q, \beta') = (vis(q), \beta')$, and we are done. ■

We now consider the complexity bound that follows from the above construction.

Theorem 6. CTL *pushdown module checking with imperfect information about the control states, but a visible pushdown store, is* 2EXPTIME*-complete.*

Proof. The lower bound follows from the known bound for *CTL* pushdown module checking with perfect information [10]. For the upper bound, Theorem 5 implies that $\mathcal{M}_S \models_r \varphi$ iff the language of the automaton $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is empty. Let $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong \rangle$ be the module induced by $\mathcal{S}$. Observe that the set of directions of the strategy trees that are the input of $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is $\mathcal{D} = \{(vis(q), \beta) : ((p,\gamma),(q,\beta)) \in \delta \text{ for some } p, q, \gamma \text{ and } \beta\}$, and it is bounded from above by $|\mathcal{S}|$. By applying Corollary 3 to $\mathcal{A}_{\mathcal{S},\neg\varphi}$ we get the required result. ■

6.2. CTL* *and* μ*-Calculus Module Checking with Visible Pushdown Store*

Let us briefly recap the approach we have taken for solving the problem in the case of *CTL*, and discuss the changes required to adapt it to other specification logics. Given an OPD $\mathcal{S}$, and a *CTL* formula φ, we build an automaton $\mathcal{A}_{\mathcal{S},\neg\varphi}$ that accepts $\{\top, \bot\}$-labeled trees corresponding to strategies ξ, whose composition with $\mathcal{M}_S$ is deadlock-free and satisfies $\neg\varphi$. Intuitively, a run of $\mathcal{A}_{\mathcal{S},\neg\varphi}$ on an input strategy tree ξ proceeds by simulating an unwinding of the module $\mathcal{M}_S$, pruned at each step according to the strategy ξ; copies of the automaton, which simulate nodes in the computation tree of $\mathcal{M}_S$ that are indistinguishable by the environment, are sent to the same direction in the input tree. The resulting run tree of $\mathcal{A}_{\mathcal{S},\neg\varphi}$ on ξ is basically a replica of the composition $\mathcal{M}_S \lhd \xi$, and the fact that it satisfies the formula φ is checked on the fly, by employing in $\mathcal{A}_{\mathcal{S},\neg\varphi}$ the classical alternating automaton (see [29]) for model checking *CTL*.

When considering *CTL** or the (propositional) μ-calculus[5], adapting the construction we used for *CTL* basically amounts to replacing the embedded alternating automaton that does the on-the-fly model checking: instead of using an automaton that handles *CTL*, one uses an automaton that handles

[5]Our construction can be extended in much the same way to the graded μ-calculus. The only subtle point involves handling the binary-encoded graded modalities without increasing the complexity with respect to the propositional μ-calculus. For more details see [5].

CTL^* or μ-calculus (see [29]). Since an alternating automaton that does μ-calculus model checking is linear in the size of the formula, while one that does CTL^* model checking is exponential in the size of the formula, the automaton $\mathcal{A}_{\mathcal{S},\neg\varphi}$ has $O(|\mathcal{S}| \cdot |\varphi|)$ states in the case of μ-calculus, and $O(|\mathcal{S}| \cdot 2^{|\varphi|})$ states in the case of CTL^*. It is important to note that the acceptance condition of $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is inherited from the embedded model checking automaton. Hence, unlike the case of CTL where a Büchi condition was enough, for the more expressive logics that we consider in this section we need the parity condition.

Theorem 7. *Consider an OPD $\mathcal{S}$ with $H_\Gamma = \emptyset$, and a* CTL* *or a propositional μ-calculus formula φ over $\mathcal{S}$'s atomic propositions. There is a PD-SPT $\mathcal{A}_{\mathcal{S},\varphi}$ such that $L(A_{\mathcal{S},\varphi})$ is exactly the set of strategies ξ for which $\mathcal{M}_S \lhd \xi$ is deadlock-free and satisfies φ. Moreover,*

- *If φ is a propositional μ-calculus formula then $\mathcal{A}_{\mathcal{S},\varphi}$ has $O(|\mathcal{S}| \cdot |\varphi|)$ states and an index $O(|\varphi|)$.*
- *If φ is a* CTL* *formula then $\mathcal{A}_{\mathcal{S},\varphi}$ has $O(|\mathcal{S}| \cdot 2^{|\varphi|})$ states and an index 3.*

Proof. We give the construction of $\mathcal{A}_{\mathcal{S},\varphi}$ for the propositional μ-calculus. The construction for CTL^* is obtained by replacing (in this or the CTL construction) the embedded classical alternating-automata model checker with a CTL^* one.

Let $\mathcal{S} = \langle AP, Q, q_0, \Gamma, \flat, \delta, \eta, Env\rangle$ be an OPD, let φ be a μ-calculus formula (guarded[6], without free variables, and in positive normal form), and let $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong\rangle$ be the module induced by $\mathcal{S}$. We build an automaton $\mathcal{A}_{\mathcal{S},\varphi}$ that accepts $\{\top, \bot\}$-labeled trees corresponding to strategies ξ, whose composition with $\mathcal{M}_S$ is deadlock-free and satisfy φ. As in [29], we are going to use a function split to avoid the problem of having states with a component in $cl(\varphi)$ that is a disjunction or a conjunction. Without the use of split, a run of the automaton may have no states that correspond to a fixpoint sub-formula of φ that is part of a conjunction or a disjunction, which makes it impossible to correctly define the acceptance condition.

The automaton $\mathcal{A}_{\mathcal{S},\varphi} = \langle\{\top, \bot\}, D, \Gamma, Q', q'_0, \flat, \delta', F\rangle$ is defined as follows

[6]The embedded μ-calculus model checking automaton requires that formulas be guarded. This guarantees that transitions involving fixpoint formulas are well defined (see [29]).

- $Q' = (Q \times (cl(\varphi) \cup \{p_\top\}) \times \{\forall, \exists\} \times \{p_e, p_s\}) \cup \{q'_0\}$.
- $\delta' : Q' \times \Sigma \times \Gamma_\flat \to \mathcal{B}^+(D \times Q' \times \Gamma_\flat^*)$. In the rules below, for the sake of succinctness, we consider $m \in \{\exists, \forall\} \times \{p_e, p_s\}$, $h \in AP \cup \{\mathbf{true}, \mathbf{false}\}$. Also, given a transition from $(\langle p, \psi, m\rangle, \top, \gamma)$, we let $p_x = p_e$ if $(p, \gamma) \in Env$, and $p_x = p_s$ otherwise.

  For all $p \in Q$, $\psi_1, \psi_2 \in cl(\varphi)$, $\psi \in cl(\varphi) \cup \{p_\top\}$, and $\gamma \in \Gamma_\flat$, we have:

  - $\delta'(q'_0, \bot, \flat) = \mathbf{false}$
  - $\delta'(q'_0, \top, \flat) = \delta'(\langle q_0, p_\top, \exists, p_s\rangle, \top, \flat) \wedge \delta'(\langle q_0, \varphi, \exists, p_s\rangle, \top, \flat)$
  - $\delta'(\langle p, \psi, \forall, p_e\rangle, \bot, \gamma) = \mathbf{true}$, and $\delta'(\langle p, \psi, \exists, p_e\rangle, \bot, \gamma) = \mathbf{false}$
  - $\delta'(\langle p, \psi, \forall, p_s\rangle, \bot, \gamma) = \delta'(\langle p, \psi, \forall, p_s\rangle, \top, \gamma)$, and $\delta'(\langle p, \psi, \exists, p_s\rangle, \bot, \gamma) = \delta'(\langle p, \psi, \exists, p_s\rangle, \top, \gamma)$
  - $\delta'(\langle p, p_\top, m\rangle, \top, \gamma) = (\bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, p_\top, \exists, p_x\rangle, \beta))$
  - $\delta'(\langle p, h, m\rangle, \top, \gamma) = \mathbf{true}$ if $h \in \eta((p, \gamma))$, or $h = \mathbf{true}$
  - $\delta'(\langle p, h, m\rangle, \top, \gamma) = \mathbf{false}$ if $h \notin \eta((p, \gamma))$, or $h = \mathbf{false}$
  - $\delta'(\langle p, \neg h, m\rangle, \top, \gamma) = \mathbf{true}$ if $h \notin \eta((p, \gamma))$, or $h = \mathbf{false}$
  - $\delta'(\langle p, \neg h, m\rangle, \top, \gamma) = \mathbf{false}$ if $h \in \eta((p, \gamma))$, or $h = \mathbf{true}$
  - $\delta'(\langle p, \psi_1 \wedge \psi_2, m\rangle, \top, \gamma) = \mathsf{split}(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \wedge \delta'(\langle p, \psi_2, m\rangle, \top, \gamma))$
  - $\delta'(\langle p, \psi_1 \vee \psi_2, m\rangle, \top, \gamma) = \mathsf{split}(\delta'(\langle p, \psi_1, m\rangle, \top, \gamma) \vee \delta'(\langle p, \psi_2, m\rangle, \top, \gamma))$
  - $\delta'(\langle p, AX\psi_1, m\rangle, \top, \gamma) = \mathsf{split}(\bigwedge_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \psi_1, \forall, p_x\rangle, \beta))$
  - $\delta'(\langle p, EX\psi_1, m\rangle, \top, \gamma) = \mathsf{split}(\bigvee_{(q,\beta)\in s(p,\gamma)}(vis(q, \beta), \langle q, \psi_1, \exists, p_x\rangle, \beta))$
  - $\delta'(\langle p, \mu y.\psi_1(y), m\rangle, \top, \gamma) = \mathsf{split}(\delta'(\langle p, \psi_1(\mu y.\psi_1(y)), m\rangle, \top, \gamma))$
  - $\delta'(\langle p, \nu y.\psi_1(y), m\rangle, \top, \gamma) = \mathsf{split}(\delta'(\langle p, \psi_1(\nu y.\psi_1(y)), m\rangle, \top, \gamma))$

  The definition of the function $\mathsf{split} : \mathcal{B}^+(D \times Q' \times \Gamma_\flat^*) \to \mathcal{B}^+(D \times Q' \times \Gamma_\flat^*)$ is a simple adaptation of the definition found in [29]. For every $d \in D, q \in Q, m \in \{\exists, \forall\} \times \{p_e, p_s\}$ and $\beta \in \Gamma_\flat^*$ we have the following:

  - $\mathsf{split}(\mathbf{true}) = \mathbf{true}$
  - $\mathsf{split}(\mathbf{false}) = \mathbf{false}$
  - $\mathsf{split}(\theta_1 \vee \theta_2) = \mathsf{split}(\theta_1) \vee \mathsf{split}(\theta_2)$

- $\mathsf{split}(\theta_1 \wedge \theta_2) = \mathsf{split}(\theta_1) \wedge \mathsf{split}(\theta_2)$
- If $\psi \in cl(\varphi)$ is of the form $p, \neg p, AX\psi', EX\psi', \mu y.\psi'(y)$ or $\nu y.\psi'(y)$, then $\mathsf{split}(d, \langle p, \psi, m\rangle, \beta) = (d, \langle p, \psi, m\rangle, \beta)$
- $\mathsf{split}(d, \langle p, \psi_1 \vee \psi_2, m\rangle, \beta) = \mathsf{split}(d, \langle p, \psi_1, m\rangle, \beta) \vee \mathsf{split}(d, \langle p, \psi_2, m\rangle, \beta)$
- $\mathsf{split}(d, \langle p, \psi_1 \wedge \psi_2, m\rangle, \beta) = \mathsf{split}(d, \langle p, \psi_1, m\rangle, \beta) \wedge \mathsf{split}(d, \langle p, \psi_2, m\rangle, \beta)$

- It remains to define the acceptance condition F. Let d be the maximal alternation level of (greatest and lowest fixpoint) sub-formulas of φ. For every $0 \leq i \leq d$, denote by G_i the set of all ν-formulas in $cl(\varphi)$ of alternation depth i, and by B_i the set of all μ-formulas in $cl(\varphi)$ of alternation depth i. Now, $F : Q' \to \{0..2d+1\}$, where:
  - For every $u \in (Q \times \{p_\top\} \times \{\forall, \exists\} \times \{p_e, p_s\}) \cup \{q'_0\}$ we have $F(u) = 0$.
  - For every $u \in (Q \times B_i \times \{\forall, \exists\} \times \{p_e, p_s\})$ we have $F(u) = 2(d - i) + 1$.
  - For every $u \in (Q \times G_i \times \{\forall, \exists\} \times \{p_e, p_s\})$ we have $F(u) = 2(d - i)$.

  Recall that, by the definition of PD-SPT, a path π of a run r is accepting iff the maximal color encountered infinitely many times along π is even. Hence, by our definition of F, such a color corresponds to the outermost fixpoint sub-formula that was visited infinitely often. Thus, the acceptance condition makes sure that the outermost fixpoint sub-formula that is visited infinitely often is a greatest fixpoint formula, and that all of its least fixpoint super-formulas are visited only finitely many times.

Note that $A_{S,\varphi}$ is semi-alternating (following the same reasoning as in Theorem 5). It is easy to see that in the construction above $\mathcal{A}_{\mathcal{S},\varphi}$ has $O(|\mathcal{S}| \cdot |\varphi|)$ states and an index $O(|\varphi|)$. For CTL^*, the embedded CTL^* model checker is of size $2^{O(|\varphi|)}$ and its index is 3 [29]. Hence, for CTL^*, $\mathcal{A}_{\mathcal{S},\varphi}$ has $O(|\mathcal{S}| \cdot 2^{|\varphi|})$ states and its index is 3. ■

We now consider the complexity bounds that follow from the above construction.

Theorem 8. *The pushdown module checking problem with imperfect information about the control states, but a visible pushdown store, is* 2EXPTIME*-complete for propositional μ-calculus specifications, and* 3EXPTIME*-complete for* CTL* *specifications.*

Proof. The lower bounds follow from the known bounds for pushdown module checking with perfect information (see [18, 19] for propositional μ-calculus, and [10] for CTL^*). For the upper bound, Theorem 7 implies that $\mathcal{M}_S \models_r \varphi$ iff the language of the automaton $\mathcal{A}_{\mathcal{S},\neg\varphi}$ is empty. Let $\mathcal{M}_S = \langle AP, W_s, W_e, w_0, R, L, \cong \rangle$ be the module induced by $\mathcal{S}$. Recall that the size of the set of directions $\mathcal{D}$ (of the strategy trees that are the input of $\mathcal{A}_{\mathcal{S},\neg\varphi}$) is bounded from above by $|\mathcal{S}|$. By applying Corollary 2 to $\mathcal{A}_{\mathcal{S},\neg\varphi}$ we get the required results. ■

7. Discussion

Recall that in our setting, whenever we push a symbol consisting entirely of invisible variables, the environment does not see the push at all. One can think of a variant of the problem where the environment does see that a push occurred, but not what was pushed. Thus, the depth of the stack is always known to the environment. It is an open question whether this variant of the problem is decidable or not.

References

[1] R. Alur, M. Benedikt, K. Etessami, P. Godefroid, T. W. Reps, and M. Yannakakis. Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27(4):786–818, 2005.

[2] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM Trans. Program. Lang. Syst.*, 23(3):273–303, 2001.

[3] B. Aminof, A. Murano, and M.Y. Vardi. Pushdown module checking with imperfect information. In *CONCUR '07*, LNCS 4703, pages 461–476. Springer-Verlag, 2007.

[4] Benjamin Aminof, Orna Kupferman, and Aniello Murano. Improved model checking of hierarchical systems. *Inf. Comput.*, 210:68–86, 2012.

[5] Benjamin Aminof, Axel Legay, Aniello Murano, and Olivier Serre. *mu*-calculus pushdown module checking with imperfect state information. In *IFIP TCS*, pages 333–348, 2008.

[6] P.A. Bonatti, C. Lutz, A. Murano, and M.Y. Vardi. The complexity of enriched μ-calculi. In *ICALP'06*, LNCS 4052, pages 540–551, 2006.

[7] Piero A. Bonatti, Carsten Lutz, A. Murano, and Moshe Y. Vardi. The complexity of enriched μ-calculi. *Logical Methods in Computer Science (LMCS 2008)*, 4(3:11):1–27, 2008.

[8] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *CONCUR'97*, LNCS 1243, pages 135–150. Springer-Verlag, 1997.

[9] L. Bozzelli, A. Murano, and A. Peron. Pushdown module checking. In *LPAR'05*, LNCS 3835, pages 504–518. Springer-Verlag, 2005.

[10] Laura Bozzelli, A. Murano, and Adriano Peron. Pushdown module checking. *Formal Methods in System Design (FMSD 2010)*, 36(1):65–95, 2010.

[11] J. Bradfield and C. Stirling. Modal μ-calculi. *Handbook of Modal Logic (Blackburn, Wolter, and van Benthem, eds.)*, pages 722–756, 2006.

[12] K. Chatterjee, L. Doyen, T. A. Henzinger, and J. Raskin. Algorithms for omega-regular games with imperfect information. In *CSL'06*, volume 4207 of *LNCS*, pages 287–302. Springer-Verlag, 2006.

[13] K. Chatterjee and T. A. Henzinger. Semiperfect-information games. In *FSTTCS'05*, volume 3821 of *LNCS*, pages 1–18. Springer-Verlag, 2005.

[14] E.M. Clarke and E.A. Emerson. Design and verification of synchronization skeletons using branching time temporal logic. In *Logics of Programs Workshop*, LNCS 131, pages 52–71. Springer-Verlag, 1981.

[15] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[16] J. Esparza, A. Kucera, and S. Schwoon. Model checking LTL with regular valuations for pushdown systems. *Inf. Comput.*, 186(2):355–376, 2003.

[17] A. Ferrante and A. Murano. Enriched μ–calculus module checking. In *FOSSACS'07*, volume 4423 of *LNCS*, page 183197, 2007.

[18] A. Ferrante, A. Murano, and M. Parente. Enriched μ–calculus pushdown module checking. In *LPAR'07*, volume 4790 of *LNAI*, pages 438–453, 2007.

[19] Alessandro Ferrante, A. Murano, and Mimmo Parente. Enriched μ-calculi module checking. *Logical Methods in Computer Science (LMCS 2008)*, 4(3:1):1–21, 2008.

[20] Eitan M. Gurari and Oscar H. Ibarra. (semi) alternating tack automata. *Theory of Computing Systems*, 15(1):211–224, 1981.

[21] D. Harel and A. Pnueli. On the development of reactive systems. In *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.

[22] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[23] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[24] D. Kozen. Results on the propositional mu–calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[25] O. Kupferman, N. Piterman, and M.Y. Vardi. Pushdown specifications. In *LPAR'02*, LNCS 2514, pages 262–277. Springer-Verlag, 2002.

[26] O. Kupferman and M. Y. Vardi. Module checking revisited. In *CAV'97*, LNCS 1254, pages 36–47. Springer-Verlag, 1997.

[27] O. Kupferman and M.Y. Vardi. Module checking. In *CAV'96*, LNCS 1102, pages 75–86. Springer-Verlag, 1996.

[28] O. Kupferman and M.Y. Vardi. Safraless decision procedures. In *IEEE FOCS'05*, pages 531–540, Pittsburgh, October 2005.

[29] O. Kupferman, M.Y. Vardi, and P. Wolper. An Automata-Theoretic Approach to Branching-Time Model Checking. *J. of ACM*, 47(2):312–360, 2000.

[30] O. Kupferman, M.Y. Vardi, and P. Wolper. Module Checking. *Information and Computation*, 164(2):322–344, 2001.

[31] R. E. Ladner, R. J. Lipton, and L. J. Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13:135 – 155, 1984.

[32] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[33] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54:267–276, 1987.

[34] G. L. Peterson and J. H. Reif. Multiple-person alternation. In *FOCS'79*, pages 348 – 363. IEEE Computer Society, 1979.

[35] J.P. Queille and J. Sifakis. Specification and verification of concurrent programs in Cesar. In *Symp. on Programming*, LNCS 137, pages 337–351. Springer-Verlag, 1981.

[36] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional μ-calculus. *Information and Computation*, 81(3):249–264, 1989.

[37] I. Walukiewicz. Pushdown processes: Games and Model Checking. In *CAV'96*, LNCS 1102, pages 62–74. Springer-Verlag, 1996.

[38] I. Walukiewicz. Model checking CTL properties of pushdown systems. In *FSTTCS'00*, LNCS 1974, pages 127–138. Springer-Verlag, 2000.